

Maestría en Inteligencia Artificial

---

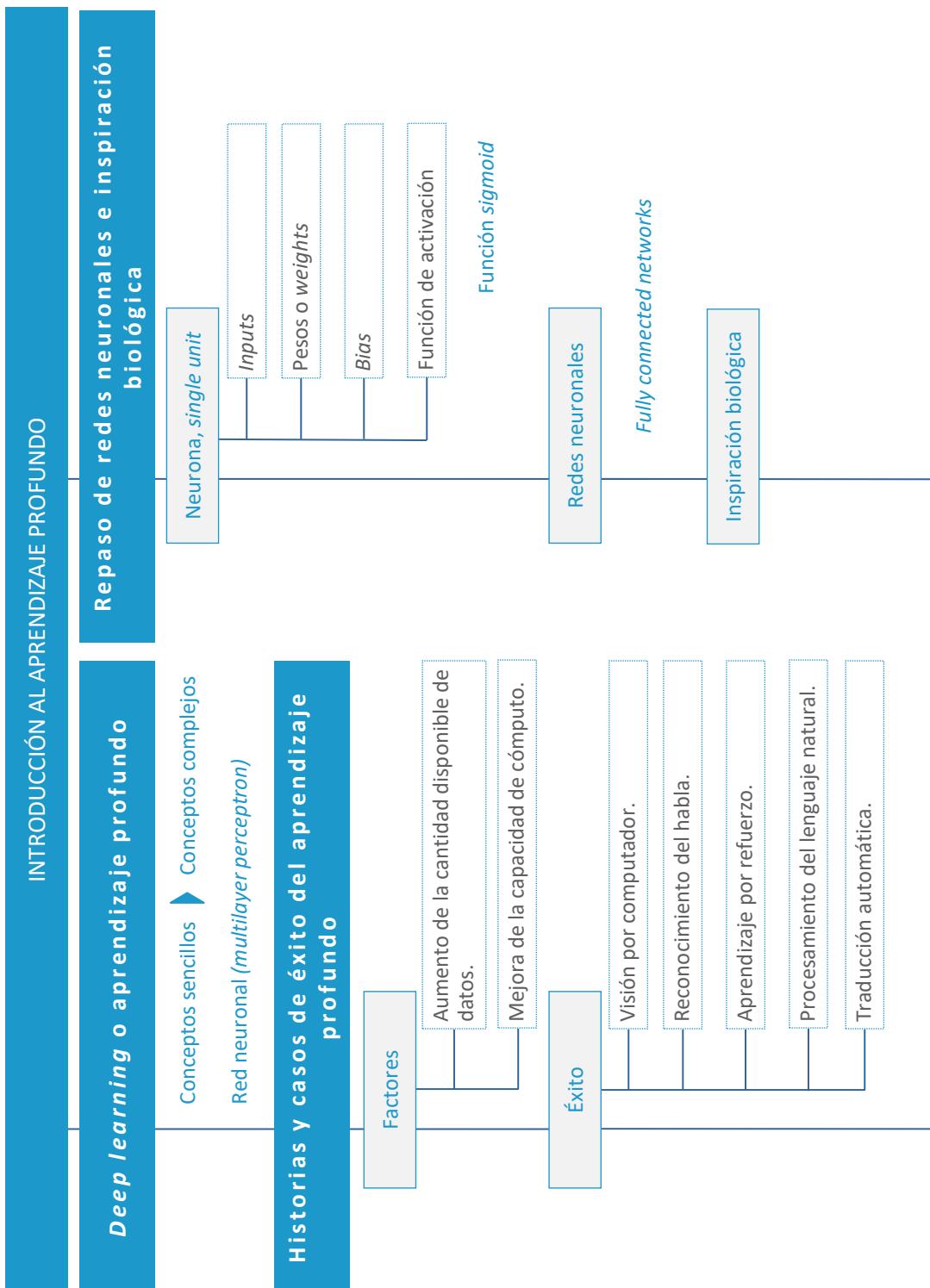
# Sistemas Cognitivos Artificiales

# Tema 1. Introducción al aprendizaje profundo

## Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
1.1. Introducción y objetivos	4
1.2. Introducción al aprendizaje profundo	5
1.3. Historia y casos de éxito del aprendizaje profundo	11
1.4. Repaso de redes neuronales e inspiración biológica	16
1.5. Referencias bibliográficas	23

# Esquema



## 1.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

Este tema servirá como introducción al aprendizaje profundo y, en general, a la asignatura. Por ello, primero estudiaremos una primera visión de qué es el aprendizaje profundo o *deep learning* y cómo encaja dentro de la inteligencia artificial (IA) y el aprendizaje automático. A continuación, comentaremos su evolución histórica y cómo este ya ha existido, en cierta manera, desde hace décadas. Finalmente, haremos un breve repaso de qué es una red neuronal y la formulación matemática de una neurona, estableciendo un paralelismo con la inspiración biológica del campo.

- ▶ En el apartado «Introducción al aprendizaje profundo» es importante relacionar dónde se encuentra el *deep learning* dentro de la IA y su relación con el aprendizaje automático que se estudió en el primer trimestre.
- ▶ En el apartado «Historia y casos de éxito del aprendizaje profundo» es importante entender de dónde vienen el aprendizaje profundo y, en especial, los factores que han llevado a su éxito, así como varias de las aplicaciones donde ha triunfado.
- ▶ Finalmente, en el último apartado, es importante comprender la formulación matemática de una neurona y recordar las distintas partes de una red neuronal, lo cual será la base de estudio del siguiente tema.

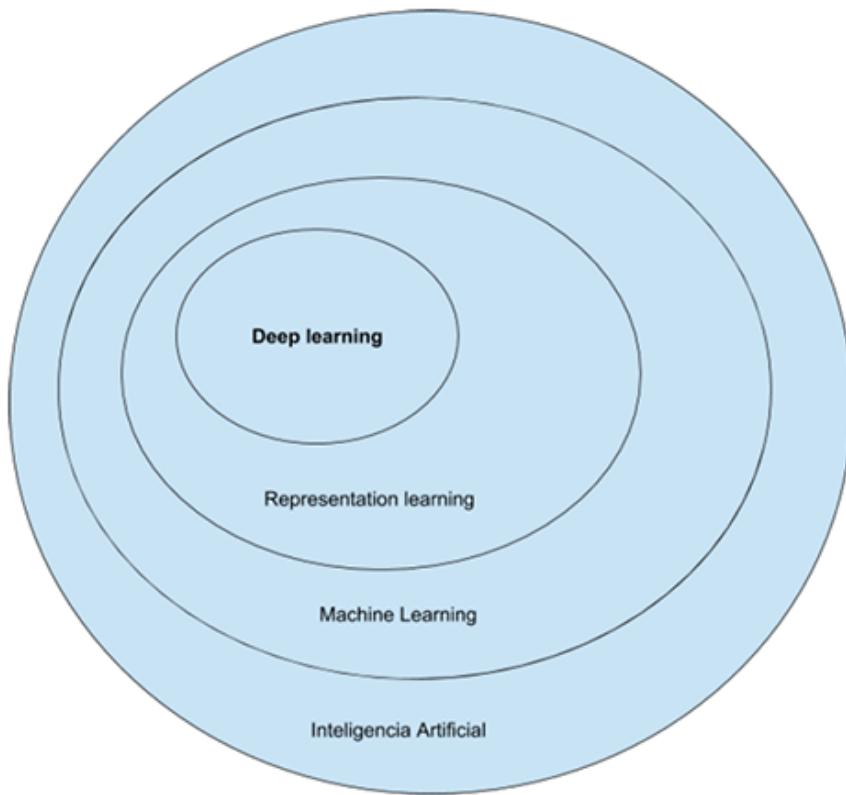


Figura 1. Gráfico de la ubicación del *deep learning* o aprendizaje profundo.

Fuente: adaptado de Goodfellow, Bengio y Courville, 2016.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 1.2. Introducción al aprendizaje profundo



Accede al vídeo «Introducción al aprendizaje profundo» a través del aula virtual

Durante años, la inteligencia artificial se fundamentó principalmente en complejos sistemas de reglas y conocimiento definidos por humanos. Estos sistemas eran capaces de resolver problemas basados en reglas formales que resultaban difíciles para las personas, pero relativamente sencillos para un ordenador. Curiosamente, las **tareas abstractas y formales** suelen ser fáciles de resolver para una máquina, mientras que las tareas más sencillas, que resultan intuitivas y directas para una

persona (como reconocer una cara) son a su vez las más difíciles de ejecutar para un ordenador.

Consideremos por ejemplo el juego del ajedrez, uno de los primeros éxitos de la IA al ser capaz de vencer a los humanos. Aunque esto fue una gran hazaña, el hecho de que el ajedrez pueda ser descrito por una serie de reglas formales en un entorno perfectamente definido facilitó su solución mediante la inteligencia artificial.

Sin embargo, mientras los ordenadores eran capaces de vencernos fácilmente al ajedrez, sus capacidades para realizar tareas casi automáticas para nosotros, como entender una frase o describir qué hay en una imagen, eran muy limitadas. Las dificultades en resolver estas tareas que presentaban los sistemas de conocimiento y de reglas provistos por humanos dio paso al *machine learning* o aprendizaje automático, una nueva área dentro de la inteligencia artificial.

**En el aprendizaje automático, los sistemas de inteligencia artificial obtienen su propio conocimiento mediante la extracción de patrones a partir de la experiencia, dada por datos.**

Por ejemplo, un sistema sencillo de aprendizaje automático puede ser entrenado mediante e-mails que son o no *spam*. A partir de los e-mails y de la clase a la que pertenezcan (ser *spam* o no), un algoritmo puede buscar automáticamente los patrones del lenguaje comunes en los distintos tipos de correo y, por tanto, «aprender» a clasificarlos.

La **representación de los datos** es crucial para el éxito de los algoritmos de *machine learning* —cosa que, curiosamente, también ocurre en la vida real: imagínese el lector el sufrimiento de hacer operaciones aritméticas en números romanos—. Muchas tareas de aprendizaje automático dependen de una buena representación de los datos; una representación que, en muchos de los casos, es de nuevo diseñada por humanos.

Por ejemplo, un sistema de *machine learning* basado en *logistic regression* es capaz de predecir si una parturienta debería recibir una cesárea dada una serie de características seleccionadas por los doctores (Mor-Yosef et al., 1990), utilizando estas características para buscar patrones de correlación con la práctica de una cesárea. Sin embargo, si este algoritmo recibiera directamente los píxeles de la imagen de resonancia magnética utilizada por el médico para obtener esas características, la capacidad de predicción sería prácticamente nula.

De este modo, muchos problemas de inteligencia artificial pueden solucionarse mediante la **definición de una serie de características en forma de datos (*features*)** que son pasadas a un algoritmo de *machine learning*. No obstante, en muchas ocasiones es complicado saber qué características deberían usarse. Por ejemplo, si queremos detectar un camión en una foto, podríamos utilizar la presencia de ruedas y de un remolque en una foto como *features*. Pero, ¿cómo describimos una rueda o un remolque en términos de píxeles? La rueda podría estar girada, tumbada o tapada; además, los camiones tienen un número distinto de ruedas; igualmente, un camión podría no llevar remolque o el remolque podría ser pequeño o muy grande... Como se ve, esta aproximación al problema se vuelve complicada y requiere de mucho esfuerzo en la definición de un conjunto adecuado de *features*, lo que se conoce como *feature engineering*.

La solución a este problema sería el área denominada ***representation learning***: que el sistema de *machine learning* no solo aprenda la solución de un problema a partir de una representación dada (*features*), sino que también aprenda la representación adecuada a partir de los datos en bruto (por ejemplo, una imagen). Las representaciones aprendidas tendrían un mejor comportamiento que las diseñadas a mano.

Sin embargo, en muchas ocasiones el problema de obtener una representación adecuada parece igual de complicado que la solución del problema en sí. Volviendo al ejemplo de reconocer camiones, podemos pensar que una buena forma de detectar un camión podría ser reconocer la presencia de una o más ruedas, de un

remolque o de otros elementos como una cabina, pero la detección de estos elementos parece un problema igual de complejo que la detección de un camión, y requiere a su vez de un nivel de comprensión casi humano. ¿Cómo diseñar un sistema que sea capaz de obtener estas representaciones de manera automática? El **deep learning, aprendizaje profundo** en español, intenta solucionar este problema mediante la creación de representaciones expresadas en términos de otras representaciones más sencillas.

**El aprendizaje profundo pretende conseguir que los ordenadores sean capaces de construir automáticamente conceptos complejos a partir de conceptos sencillos.**

El ejemplo por excelencia de un modelo de *deep learning* es la **red neuronal** o perceptrón multicapa (*multilayer perceptron*). Una red neuronal es básicamente una función matemática definida por nodos en capas donde cada nodo es una función más simple que depende de los nodos de la capa anterior. Puede pensarse, pues, que cada nodo utiliza los conceptos simples aprendidos en capas anteriores para desarrollar un concepto más complejo a partir de ellos.

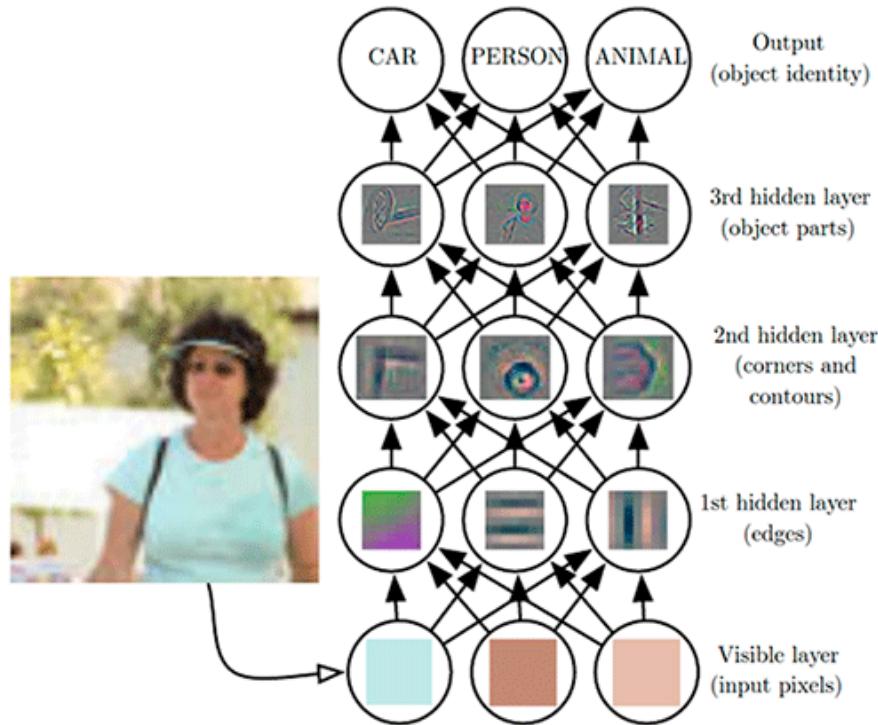


Figura 2. Ejemplo de una red neuronal aprendiendo conceptos complejos.

Fuente: Goodfellow, Bengio y Courville, 2016.

En la imagen anterior (figura 2) puede verse un ejemplo de cómo las redes neuronales aprenden conceptos más abstractos y complejos, aprendiendo efectivamente una representación de los datos adecuada. Resolver el problema en la figura (identificar a una persona, coche o animal) con una función directa a partir de los píxeles de la imagen sería casi imposible. Una red neuronal divide la tarea en una serie de problemas más sencillos a través de sus distintas capas. En el ejemplo, ante la presencia de imágenes:

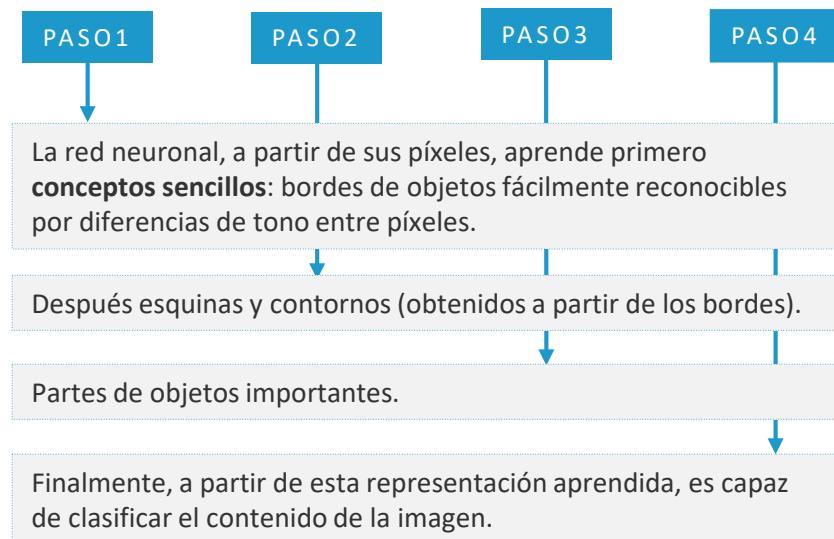


Figura 3. Ejemplo de cómo una red neuronal aprende de conceptos sencillos a complejos.

Esta jerarquía de conceptos aprendida por la red neuronal nos puede ayudar a entender el porqué del nombre de «*deep* o profundo». Para obtener una representación exitosa del problema, se necesita una jerarquía grande o profunda de conceptos. Tradicionalmente se ha asociado esta profundidad con el número de capas en una red neuronal, sin embargo, no hay un número a partir del cual una red neuronal sea considerada «profunda», aunque muchas de las arquitecturas más famosas se han caracterizado por añadir más y más capas.

En resumen, el aprendizaje profundo o *deep learning* es un tipo de aprendizaje automático o *machine learning*, que a su vez es un área dentro de la inteligencia artificial. Dentro del aprendizaje automático, el aprendizaje profundo se basa en la representación del problema a tratar en una jerarquía de conceptos o abstracciones cada vez más complejos y obtenidos a partir de conceptos más sencillos.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 1.3. Historia y casos de éxito del aprendizaje profundo



Accede al vídeo «Historia y casos de éxito del aprendizaje profundo» a través del aula virtual

Si bien el aprendizaje profundo está viviendo su edad dorada ahora mismo, representando los mayores avances de la inteligencia artificial en los últimos tiempos, es importante resaltar que no es un área de estudio nuevo, sino que ha existido en la literatura con distintos nombres ya desde hace varias décadas. Diversos factores han contribuido a su resurgimiento estos últimos años, en especial la disponibilidad de muchos más datos para entrenar modelos y el aumento de la capacidad de cómputo.

### Breve historia del *deep learning*

El aprendizaje profundo ha tenido varios nombres a lo largo de la historia, incluyendo *cybernetics*, conexiónismo o simplemente redes neuronales artificiales. La inspiración biológica ha sido una constante en el campo, partiendo de la idea de que el cerebro es un ejemplo de aprendizaje e inteligencia: si podemos replicar su funcionamiento con algoritmos, podremos obtener sistemas inteligentes. Si bien la inspiración está ahí, lo cierto es que en la actualidad el campo no está guiado por la neurociencia, nuestra comprensión del cerebro no es lo suficientemente avanzada.

### Primeros modelos

La neurona de McCulloch-Pitts (originaria de 1943) y el **perceptrón** (1958) son los primeros ejemplos de modelos que, de una manera muy simplificada, están inspirados en el funcionamiento de una neurona aislada. Del mismo modo, el sistema **ADALINE**, *Adaptive Linear Element* (1960), introdujo un sistema de entrenamiento

muy parecido al *stochastic gradient descent* que se utiliza ahora para entrenar redes neuronales.

## Años 80

En los años 80 surgió una nueva ola de interés en redes neuronales, principalmente gracias a un movimiento interdisciplinar llamado conexiónismo, bajo el área de la ciencia cognitiva. La idea central del **conexiónismo** es que un gran número de pequeñas unidades de cómputo puede alcanzar un comportamiento inteligente mediante su conexión en una red. La idea del algoritmo de *back-propagation*, clave en el entrenamiento de redes neuronales, data de esta época, así como la idea de las representaciones distribuidas, ambos conceptos que veremos durante el curso.

## Años 90

Nuevos avances se produjeron en el área de problemas con secuencias. Las redes neuronales secuenciales LSTM, muy utilizadas en la actualidad, datan por ejemplo de 1997 (Hochreiter y Schmidhuber, 1997). Sin embargo, en esta época el interés por las redes neuronales volvió a decaer ante la dificultad de obtener buenos resultados. Otros algoritmos y técnicas de aprendizaje automático, como las *support vector machines*, coparon durante varios años el interés de la comunidad, convirtiéndose en las soluciones estándar a la mayor parte de los problemas tratables por el *machine learning*.

## A partir de 2006

Sin embargo, a partir de 2006, varios avances conseguidos por investigadores como Geoffrey Hinton, Yoshua Bengio y Yann LeCun volvieron a situar a las redes neuronales en el centro del tablero. Año a año, estas empezaron a superar en muchas áreas a las otras técnicas de aprendizaje profundo, en ocasiones por un gran margen, y consiguieron resolver problemas que se pensaba llevaría años o décadas solucionar. El hecho de que fuera posible entrenar con éxito redes más profundas (con más

capas) llevó poco a poco al famoso nombre actual de *deep learning*. Sin embargo, ¿cómo es posible que, cuando gran parte de los algoritmos y las técnicas de entrenamiento ya estaban inventados, llevara tanto tiempo al aprendizaje profundo a afianzarse como una parte fundamental y disruptiva del mundo del *machine learning*?

## Factores del éxito del *deep learning*

Si bien las redes neuronales eran ya usadas con éxito antes de la explosión del *deep learning*, se consideraba en general que eran muy complicadas de entrenar. Dos factores pueden explicar su resurgimiento en los últimos años: la disponibilidad de muchos más datos y la mejora de la capacidad de cómputo.

El **aumento de la cantidad disponible de datos** en una sociedad tan digital como la nuestra ha permitido contar con más recursos para entrenar los algoritmos de *machine learning*. En los años 80 y 90 los investigadores utilizaban datasets de centenares o varios miles de puntos. En la actualidad, los datasets en los que el aprendizaje profundo está demostrando sus capacidades llegan a constar de millones de puntos. Las redes neuronales son algoritmos complejos, y una de las razones por las que podían no estar alcanzando todo su potencial era por no contar con la cantidad adecuada de datos para su entrenamiento.

Por otro lado, la **mejora de la capacidad de cómputo**, inherente al avance de la informática, ha permitido que la experimentación requiera menos tiempo y que se puedan ejecutar modelos más complicados, así como modelos con más datos. Las redes neuronales con mayor número de neuronas son capaces de solucionar problemas más complejos y desarrollar mejores representaciones de los datos brutos. De manera similar, otros avances en *hardware* como la utilización de tarjetas gráficas para el entrenamiento de redes neuronales han ayudado en gran medida al desarrollo del campo.

## Casos de éxito del *deep learning*

El éxito del aprendizaje profundo en la actualidad ha hecho que cada vez se aplique a más problemas, incluso en campos distintos de donde las soluciones de inteligencia artificial se han aplicado tradicionalmente. Sin embargo, hay una serie de casos especiales donde el *deep learning* ha mejorado notablemente, incluso podría decirse revolucionado los resultados, contribuyendo a su éxito presente.

Probablemente el área más famosa donde el aprendizaje profundo ha funcionado de manera espectacular sea la de **visión por computador** (*computer vision*) y, en particular, el reconocimiento de objetos. En 2012, una *convolutional neural network* ganó por primera vez y con gran margen la competición de reconocimiento de objetos ImageNet, consistente en la clasificación de 1000 objetos en un conjunto de más de un millón de imágenes. Durante los años siguientes, nuevas redes profundas aumentaron enormemente la capacidad de clasificar imágenes hasta el punto de que el problema se considera prácticamente resuelto. Otras áreas de la visión por computador donde el *deep learning* ha tenido gran éxito son la detección de objetos en imágenes y la segmentación.

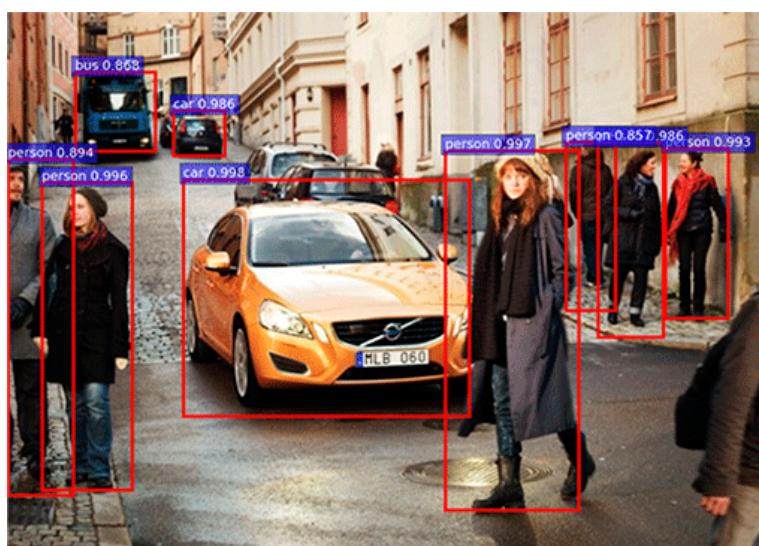


Figura 4. Ejemplo de visión por computador con *deep learning*.

Fuente: <https://bigsnarf.wordpress.com/2016/11/07/faster-r-cnn-pedestrian-and-car-detection/>

En la figura 4 vemos que los sistemas de detección de peatones, desarrollados con tecnologías de visión por computador con *deep learning*, son clave para las nuevas tecnologías de coches sin conductor.

Otra área donde el *deep learning* ha mejorado ostensiblemente los resultados anteriores es el del **reconocimiento de habla** (*speech recognition*). Prácticamente todos los asistentes digitales que utilizamos en nuestros móviles están programados mediante redes neuronales para reconocer lo que decimos.

Asimismo, el área de **aprendizaje por refuerzo** (*reinforcement learning*) ha visto también grandes mejoras de la mano del aprendizaje profundo. DeepMind, parte de Alphabet, demostró cómo un sistema de *reinforcement learning* puede aprender a jugar a videojuegos de Atari directamente a partir de las partidas. En un hito para la historia de la inteligencia artificial, otro sistema desarrollado por esta compañía consiguió derrotar al campeón mundial de *go* en 2016. El *go*, un juego tremadamente complejo, no se consideraba un candidato a ser resuelto por técnicas de inteligencia artificial en el futuro inmediato.

Por último, el **procesamiento del lenguaje natural** (*natural language processing*) es otra área en la que, aunque de manera un poco más tardía, el aprendizaje profundo está empezando a dar buenos resultados. Problemas clásicos como *part of speech tagging* o análisis de sentimientos han obtenido nuevas soluciones de vanguardia basadas en redes neuronales. Igualmente, los sistemas de **traducción automática** como el archiconocido Google Translate se ejecutan ahora sobre soluciones de aprendizaje profundo.

Es importante mencionar que, aunque el *deep learning* está revolucionando la inteligencia artificial y ha adquirido una gran presencia hasta en la prensa y el mundo de los negocios, este no es más que otra técnica dentro del mundo del aprendizaje automático. No todos los problemas son solucionados de manera óptima por redes profundas. Otros algoritmos ofrecen soluciones a veces más efectivas, sencillas y rápidas, si bien es cierto que las redes neuronales profundas son probablemente la

herramienta más versátil y potente en el campo de la inteligencia artificial en la actualidad, esto no quiere decir que el resto hayan quedado automáticamente desfasadas o que incluso en unos años no se encuentre una nueva técnica que adquiera todo el protagonismo.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 1.4. Repaso de redes neuronales e inspiración biológica



Accede al vídeo «Repaso de redes neuronales e inspiración biológica» a través del aula virtual

En este apartado repasaremos las nociones básicas de redes neuronales que ya conocemos de la asignatura de *Aprendizaje Automático*. Nótese que, a partir de aquí, la terminología técnica será escrita mayoritariamente en inglés, ya que este es el idioma que se utiliza normalmente en la literatura y la traducción de muchos términos no está estandarizada, al igual que hace más difícil la búsqueda en Internet de recursos relacionados.

### Neurona, *single unit*

Recordemos primero cómo funciona una simple neurona. Tal y como vemos en la figura 5, una neurona o nodo en una red neuronal consiste en:

- ▶ Una serie de **inputs** ( $a_1, a_2, a_3 \dots a_n$  en el diagrama).
- ▶ Una serie de pesos o **weights** ( $w$  en el diagrama).

- ▶ Una función de activación ( $\sigma$ ), que se aplica sobre la suma del *bias* con el producto de cada *input* por su peso correspondiente.

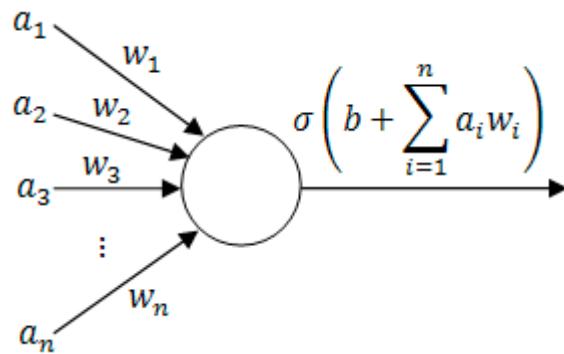


Figura 5. Expresión matemática de una neurona en una red neuronal.

Fuente: <https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc>

La función de activación se conoce también como ***non-linearity***. Durante el curso, veremos varias funciones que pueden ser utilizadas aquí; una de las opciones más tradicionales, que asemeja a una neurona al algoritmo de *logistic regression*, es la **función *sigmoid*** (tradicionalmente escrita, de hecho, como  $\sigma$ ).

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

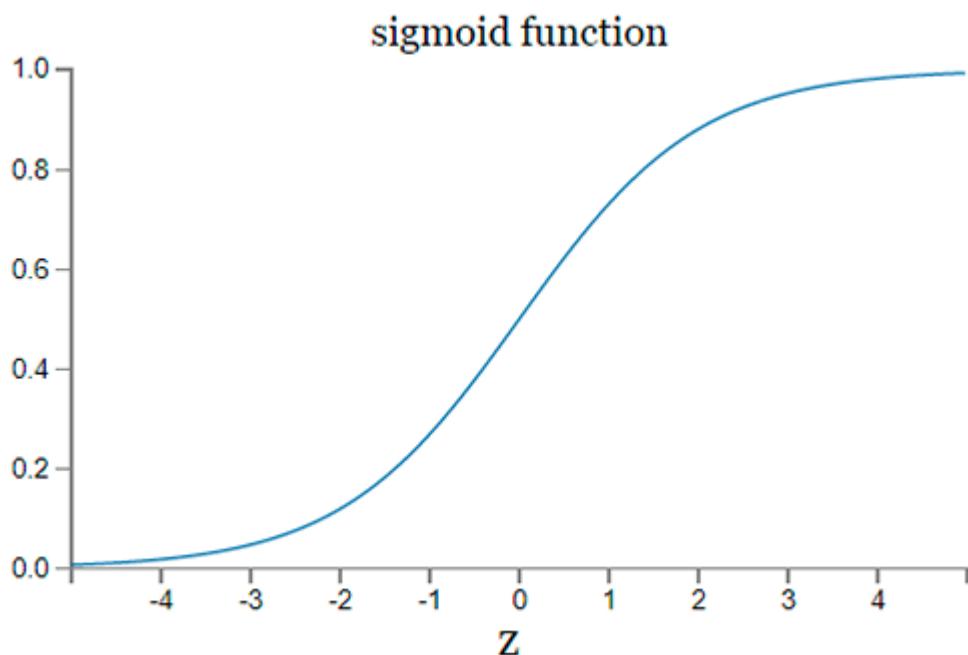


Figura 6. Función *sigmoid*.

Fuente: <http://neuralnetworksanddeeplearning.com/chap1.html>

Como podemos comprobar, la función toma valores entre 0 y 1: tiende a 1 para valores cercanos a infinito y a 0 para valores cercanos a menos infinito. Es por eso que esta función se utiliza normalmente para modelar probabilidades.

Intentemos intuir el funcionamiento de una neurona, en este caso como si de un algoritmo aislado se tratara o, visto de otro modo, como si se tratara de un simple algoritmo de *logistic regression*.

**Ejemplo:**

Supongamos que, en un ejemplo muy sencillo, queremos que el *output* de nuestra neurona modele la probabilidad de que nos vayamos de vacaciones.

Digamos que las *features* que seleccionamos para modelar este problema como las inputs de la neurona son:

$a_1$  = cantidad de dinero en el banco en miles de euros (para nuestro ejemplo, 5).

$a_2$  = número de amigos que nos acompañan (para nuestro ejemplo, 2)

$a_3$  = horas de trabajo pendiente (para nuestro ejemplo, 1).

Como vemos en la fórmula, cada *input* es multiplicado por su propio *weight*. Podemos entender los pesos o *weights* como cuánto ponderamos ese *input* en la salida final. Asumamos que, en nuestro caso particular, esta **ponderación** es:

$$w_1 = 2$$

$$w_2 = 4$$

$$w_3 = -3$$

En el caso particular del ejemplo descrito, el valor que recibe la **función de activación** sería:  $2 \cdot 5 + 4 \cdot 2 - 3 \cdot 1 = 15$ , con  $\sigma(15)$  valiendo prácticamente 1.

De modo que, para los pesos que modelan nuestro comportamiento y ante los *inputs* de nuestra situación particular, obtenemos que la neurona nos asigna una probabilidad casi 1 de irnos de vacaciones.

No hemos hablado del *bias*. Este término, si bien no es totalmente imprescindible para entender el funcionamiento de una neurona, representa un **valor constante** a tener en cuenta que forma parte del modelado del problema. En nuestro ejemplo, podríamos entenderlo como un valor inherente al modelo que define nuestra predisposición a irnos o no de vacaciones: supongamos que, en general, nos encanta irnos de vacaciones en todo momento independientemente de las condiciones. En ese caso, el *bias* ( $b$ ) sería un valor positivo que ayudaría a llevar la probabilidad de irnos de vacaciones a valores más altos; en términos estadísticos tendríamos un *bias* (una predisposición) a irnos de vacaciones.

**De este modo, los *bias* permiten a una neurona modelar un valor constante e independiente de los *inputs* que influye en la salida de la misma.**

Este ejemplo nos da una noción de cómo funciona una neurona por medio de la ponderación de sus *inputs* y de asignar un *output* o valor de salida. Como vemos, a

valores positivos obtenemos salidas cercanas a 1, mientras que a valores negativos, salidas cercanas a 0. Por supuesto, la idea aquí es que los pesos ( $w$ ) y el *bias* ( $b$ ) sean aprendidos a partir de los datos y no tengan que conocerse de antemano.

Podríamos recolectar los datos de momentos en los que nos fuimos o no de vacaciones en el pasado y permitir que la neurona asignara esos pesos de manera automática. Es importante destacar también que, aunque en este caso la salida de la neurona puede entenderse como una probabilidad, esto no es una norma. De hecho, veremos funciones de activación distintas de *sigmoid* con valores fuera del intervalo  $[0, 1]$ .

## Redes neuronales

Una sola neurona representa una **función lineal**. Si bien los modelos lineales son bastante poderosos, tienen ciertas limitaciones, la más clara de ellas es que no permiten la interacción de distintos *inputs*. Como se ve en la ecuación, cada input está multiplicado por su respectivo peso, pero no interactúan entre ellos.

En la introducción histórica ya vimos que la corriente conexionista defendía que un gran número de pequeñas unidades de cómputo pueden alcanzar un comportamiento inteligente mediante su conexión en una red. De manera similar, juntando varias neuronas podemos crear una red neuronal, capaz de resolver problemas más complejos y de permitir interacciones entre los diversos *inputs* mediante la conexión de distintas neuronas.

**Una red neuronal, también llamada *multilayer perceptron* (MLP), es un conjunto de neuronas simples situadas en capas.**

Normalmente, al hablar de redes neuronales nos referimos a *fully connected networks*: redes donde todas las neuronas o nodos de una capa están conectados con todas las neuronas de la capa anterior.

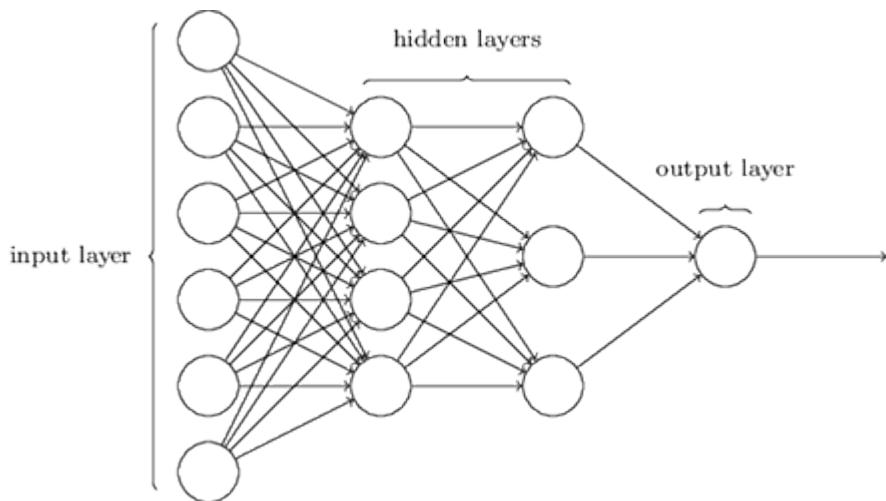


Figura 7. Red neuronal.

Fuente: <http://neuralnetworksanddeeplearning.com/chap1.html>

Como se ve en imagen anterior (figura 7), las distintas neuronas se colocan en capas y reciben como *inputs* la salida de todas las neuronas de la capa anterior. Capa por capa, los *outputs* de una neurona son una fuente de datos para las neuronas de la siguiente capa. Vemos aquí claramente la idea clave del *deep learning*: obtener representaciones cada vez más complejas y basadas en conceptos simples.

Una red neuronal tiene varios tipos distintos de capas:

- ▶ La primera capa, la *input layer*, es la capa que recoge los datos a tratar.
- ▶ La última capa, *output layer*, representa la salida de la red.
- ▶ Finalmente, las capas interiores se denominan *hidden layers*.

**Entrenar una red neuronal** significa encontrar los valores óptimos de los pesos ( $w$ ) y *bias* ( $b$ ) de todas las neuronas que mejor describen los datos en términos del *output* deseado. Por ejemplo, y como ya hemos visto: el *output* podría ser la probabilidad de que una imagen sea de un camión y el *input*, todos los píxeles que componen esa imagen.

Otro punto a tratar aquí es que una red del tipo de la figura 7 se denomina *feed-forward network*. Esto es debido a que no hay bucles en la red y la información

siempre fluye hacia delante. Durante el curso, veremos otro tipo de redes que no siguen este paradigma, llamadas *recurrent neural networks*.

## Inspiración biológica

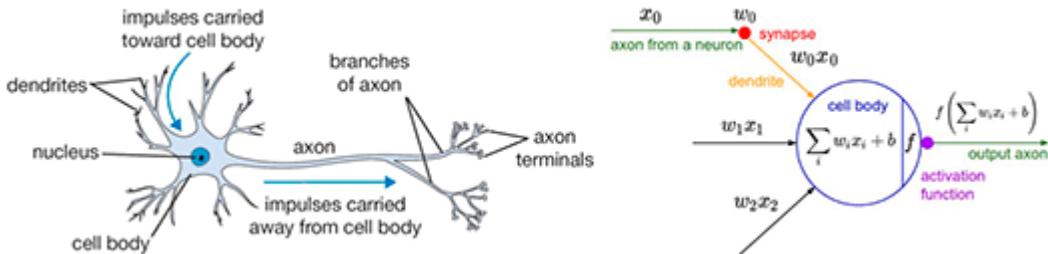


Figura 8. Neurona real y neurona de una red neuronal.

Fuente: <https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc>

La inspiración biológica de las redes neuronales es clara como vemos en la figura 8. Una neurona común obtiene impulsos de otras neuronas a través de las dendritas y transmite a su vez un impulso a otras neuronas a través del axón. Esto es equivalente a los *inputs* y *outputs* que hemos visto. De manera similar al cerebro, las redes neuronales se componen de muchas neuronas y sus conexiones, pero es importante volver a recalcar que las similitudes acaban pronto.

La realidad es que hay muchos tipos distintos de neuronas en el cerebro y que estas no actúan con las funciones de activación o las arquitecturas de red normalmente utilizadas en el campo del *deep learning*. Por ello, pese a que la neurociencia ha servido de clara inspiración al campo del aprendizaje profundo, ambas disciplinas científicas siguen caminos y objetivos separados.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 1.5. Referencias bibliográficas

Goodfellow, I., Bengio, Y. y Courville, A. (2016). *The Deep Learning Book*. Cambridge (Estados Unidos): The MIT Press.

Hochreiter, S. y Schmidhuber, J. (1997). Long Short-Term Memory. *Journal Neural Computation*, 9(8), 1735-1780.

Mor-Yosef, S. et al. (1990). Vaginal Delivery following One Previous Cesarean Birth: Nation Wide Survey. *The Journal of Obstetrics and Gynaecology Research*, 16(1), 33-37. Recuperado de <https://obgyn.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1447-0756.1990.tb00212.x>



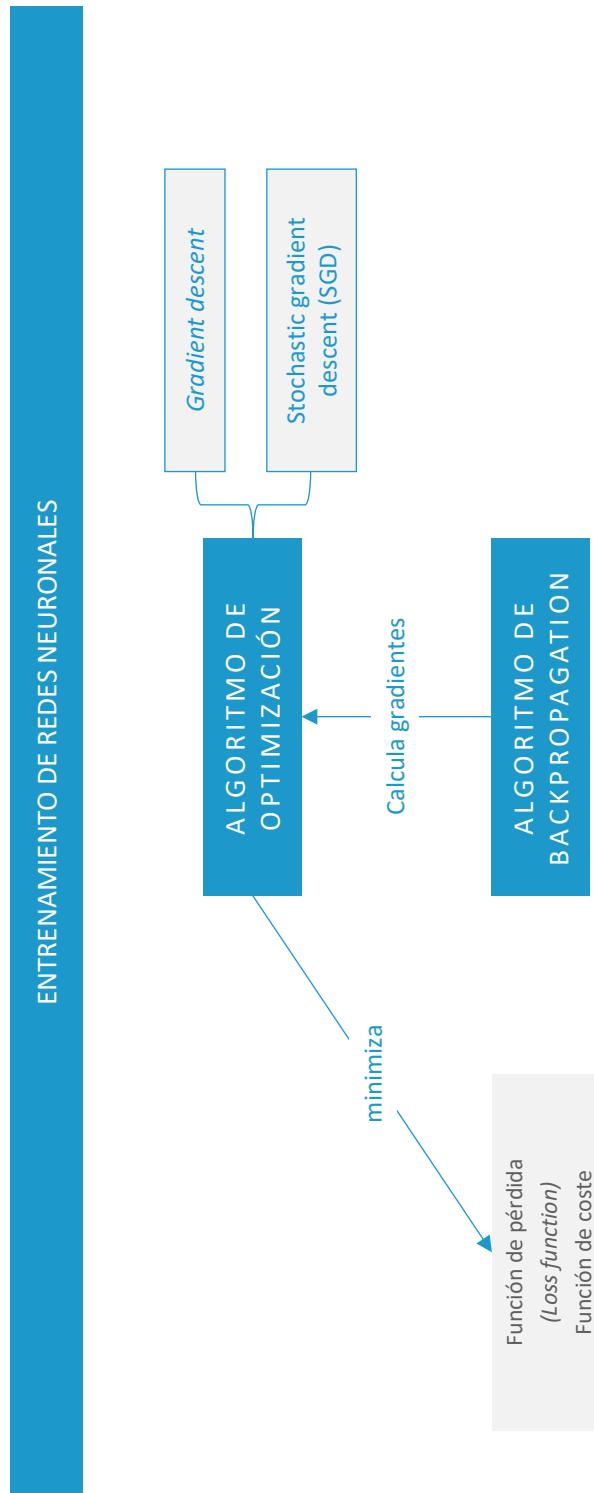
Accede al vídeo «Resumen» a través del aula virtual

# Tema 2. Entrenamiento de redes neuronales

## Índice

Esquema	25
Ideas clave	26
2.1. Introducción y objetivos	26
2.2. Funciones de coste	27
2.3. Entrenamiento con <i>gradient descent</i>	32
2.4. <i>Backpropagation</i>	38

# Esquema



## 2.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

Trataremos varios conceptos fundamentales en el mundo de las redes neuronales y será el tema con más carga matemática del curso. Es importante comprender los conceptos que se ven aquí: función de coste, entrenamiento como problema de optimización, *gradient descent*, *backpropagation*... y cómo se relacionan entre ellos para dar lugar al entrenamiento de redes neuronales.

El contenido se estructura de la siguiente manera:

- ▶ Para empezar, veremos qué es una **función de coste** y un ejemplo de red neuronal utilizado para clasificar imágenes.
- ▶ A partir del concepto de función de coste, aprenderemos cómo entrenar una red neuronal a partir del problema de optimización de minimizar dicha función.
- ▶ Para ello, estudiaremos el **algoritmo de gradient descent** y la versión utilizada en la práctica, *stochastic gradient descent*.
- ▶ Finalmente, veremos cómo calcular los gradientes de una red neuronal de manera eficiente con el **algoritmo de backpropagation**.

Si bien no es necesario ser capaz de desarrollar de manera completa el algoritmo de *backpropagation* para una red neuronal, es muy importante entender su funcionamiento y ser capaz de desarrollar ejemplos relativamente sencillos como los vistos en clase. Este algoritmo es fundamental para entender conceptos que veremos más adelante durante el curso.

Se recomienda encarecidamente ver los vídeos mostrados en la sección «Lo + recomendado» como soporte para comprender este tema y entender el funcionamiento de las redes neuronales.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 2.2. Funciones de coste



Accede al vídeo «Funciones de coste» a través del aula virtual

### Una red neuronal para MNIST

Como hemos indicado, en este tema veremos los fundamentos matemáticos de las redes neuronales y cómo se entrena una red neuronal. Para ello, utilizaremos un problema clásico en el mundo del *machine learning*: MNIST.

MNIST es un dataset de imágenes con números escritos a mano del 1 al 10, siendo el objetivo detectarlos automáticamente.

Si bien ya desde los años 90 existen sistemas basados en redes neuronales entrenadas con este dataset (por ejemplo, para detectar códigos postales en el correo), MNIST sigue siendo utilizado como un *benchmark* para algoritmos de visión por computador. Podría decirse que es una especie de «Hola, mundo» en el mundo del aprendizaje automático.

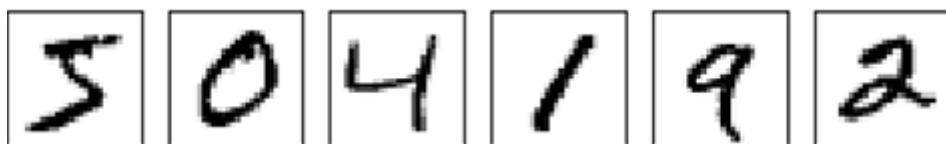


Figura 9. Ejemplo de imágenes en MNIST representando números.

Fuente: <http://neuralnetworksanddeeplearning.com/chap1.html>

Para resolver el problema de clasificar dígitos utilizaremos, como no podía ser de otra manera, una red neuronal.

1. Ya que las imágenes están compuestas de píxeles, alimentaremos a la red con todos los píxeles de la imagen representados en un vector de una dimensión; para ello, se concatenan todas las filas de píxeles de la imagen, una tras otra.
2. Como las imágenes de MNIST son de 28x28 píxeles (muy baja resolución), la *input layer* tendrá 784 neuronas. Cada neurona de esta primera capa representará un píxel, con un valor entre 0 y 1 según la oscuridad del píxel, siendo 0 totalmente blanco y 1 totalmente negro.
3. Para este ejemplo, utilizaremos una sola *hidden layer*, aunque podríamos añadir más si quisieramos. El número de nodos en esta capa es algo con lo que se puede experimentar, añadiendo o quitando potencia a la red.
4. Finalmente, la *output layer* tendrá 10 nodos, uno por cada posible número a clasificar. Si utilizamos la función de activación *sigmoid* (que vimos en el tema anterior) en cada uno de estos nodos de salida, podemos entender que cada neurona de salida representa la probabilidad de que la imagen sea un número en particular. De este modo, podremos clasificar la imagen en un número con base en el mayor valor obtenido en esta última capa.

En la figura 3 vemos un esquema de esta red neuronal.

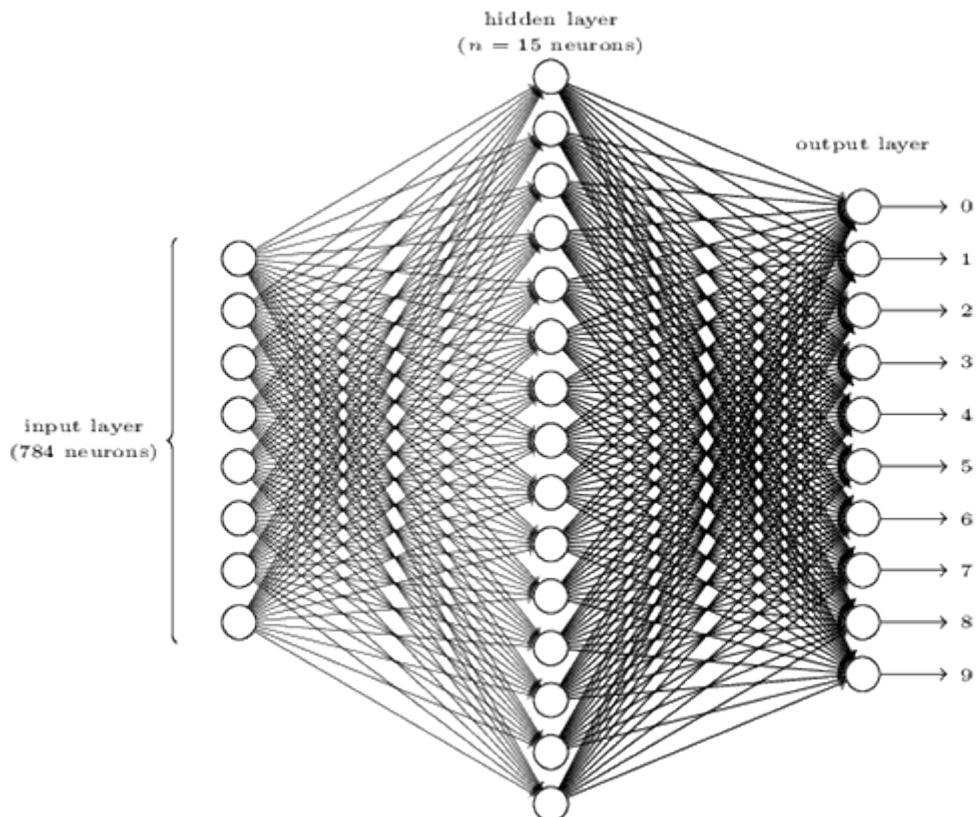


Figura 10. Red neuronal para MNIST.

Fuente: <http://neuralnetworksanddeeplearning.com/chap1.html>

Una pregunta que podríamos hacernos es cuántas neuronas tenemos que tener en la *hidden layer*. Este número es un **hiperparámetro** (*hyperparameter*) de la red y tenemos libertad para elegirlo. Una forma de obtenerlo sería mediante una búsqueda de hiperparámetros, probando varios valores y quedándonos con el que ofrece mejores resultados.

Es importante obtener cierta idea de lo que puede estar pasando en esta capa intermedia. Algo que debería quedar claro es que, al añadir más nodos a la *hidden layer*, estamos añadiendo más potencia a la red, ya que a partir de los píxeles de entrada obtenemos **más representaciones intermedias** de los elementos en una imagen (recordemos, las redes neuronales obtienen conceptos más complejos a partir de conceptos simples). ¿Qué representaciones intermedias están obteniendo estas neuronas a partir de los píxeles? Si bien esto variará cada vez que entrenemos la red neuronal, la red intentará representar de la mejor manera posible con los recursos que tiene el problema a tratar. Por ejemplo, algo que podría estar pasando

es que las neuronas de la *hidden layer* se activen cuando ven ciertos píxeles negros en la entrada para reconocer formas sencillas, como se representa en la figura 4. Así, las neuronas intermedias se activarían al reconocer formas simples, las cuales pueden ser utilizadas a su vez por la última capa para activar las neuronas de salida asociadas a números que están compuestos por esas formas.

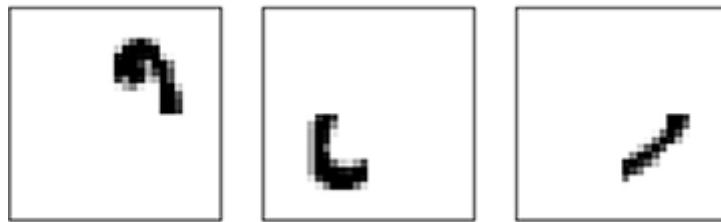


Figura 11. Posibles representaciones intermedias en la *hidden layer*.

Fuente: <http://neuralnetworksanddeeplearning.com/chap1.html>

Es importante mencionar que, si bien en este caso podríamos obtener una visualización de lo que está pasando en la capa intermedia, en general las abstracciones que la red neuronal obtiene al ser entrenada pueden escapar completamente a nuestra comprensión. Al fin y al cabo, las redes neuronales son potentes modelos estadísticos capaces de encontrar patrones muy complejos en los datos.

Volviendo a la pregunta de cuántas neuronas necesitamos en la capa intermedia, podemos ya intuir que no hay una respuesta clara. Necesitamos un número suficiente para que la red neuronal obtenga unas representaciones intermedias adecuadas: con pocas neuronas, la red no obtendrá representaciones suficientemente complejas y su precisión se verá afectada. Con más neuronas, llegará un punto en el que estas no ayudarán, ya que la red tiene suficiente capacidad para expresar la variabilidad encontrada en los datos.

## Definiendo una función de coste

Empecemos a introducir cierta notación matemática sobre **cuál es nuestro objetivo** al entrenar una red neuronal. Definiremos como:

- ▶  $x$  = el *input* o *training example* de la red (en este caso, nuestro vector de 784 píxeles).
- ▶  $f(x)$  = el valor deseado de salida de la red, representado como un vector con 10 componentes, uno por cada número.

Por ejemplo, si  $x$  representa una imagen con un 3, entonces:

$$f(x) = (0,0,0,1,0,0,0,0,0,0)$$

$f(x)$  representa la función real que queremos imitar, una función que a cada imagen asigna el número representado. Nuestra red neuronal también es una función que a partir de  $x$ , de los pesos  $w$  y *biases*  $b$  obtiene a su vez otro vector con 10 componentes. La definiremos como  $a(x, w, b)$  o solo  $a$  para simplificar.

Nuestro objetivo es obtener los parámetros  $w$  y  $b$  de la red neuronal que mejor aproximen la función real  $y(x)$  para todos los valores  $x$ , esto es, para todos los *training examples* en nuestro dataset. Para cuantificar cómo de buena es esta aproximación, definimos la función de coste (*cost function*):

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

Donde:

- ▶  $w$  y  $b$  representan todos los parámetros de la red neuronal.
- ▶  $n$  es el número de *data points* en nuestro dataset.
- ▶ El sumatorio es sobre todas las imágenes del dataset.

Como podemos comprobar, por cada ejemplo estamos obteniendo el cuadrado de la distancia vectorial, un número siempre positivo, entre el valor real y la salida de nuestra red. Para una imagen que nuestra red clasifica de manera correcta como un 0, podríamos tener  $a(x) = (0.9, 0.1, 0.1, 0.2, 0.1, 0.1, 0, 0, 0, 0)$ , que como vector está cerca de  $f(x) = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$ . De este modo, podemos ver que la

función de coste es una **suma de valores que miden el error** que nuestra red está obteniendo al clasificar las imágenes.

Si  $C(w, b)$  es muy grande, nuestra red no está haciendo un buen trabajo. Por lo tanto, nuestro objetivo al entrenar la red será minimizar este error y acercar en la medida de lo posible  $C(w, b)$  a 0. Al final, y como es común en el mundo de *machine learning*, el problema a resolver es realmente un **problema de optimización**. El objetivo es obtener los parámetros  $w$  y  $b$  que minimicen el valor de  $C$ .

Podríamos preguntarnos por qué el error se mide con el cuadrado de la distancia entre vectores. Este tipo de error es, de hecho, muy típico en problemas de aprendizaje automático y se conoce como **MSE (Mean Squared Error)**. ¿Por qué no, entonces, minimizar directamente el número de fallos cometidos por la red? La respuesta es que necesitamos una función diferenciable para poder resolver el problema de optimización. Como veremos próximamente, el algoritmo que utilizaremos para minimizar la función, *gradient descent*, necesita una función de coste diferenciable.

Es importante mencionar que el *mean squared error* no es la única manera de medir el error. Hay otras funciones posibles que derivan en diferentes problemas de optimización y, por tanto, en diferentes resultados a la hora de entrenar.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 2.3. Entrenamiento con *gradient descent*



Accede al vídeo «Entrenamiento con *gradient descent*» a través del aula virtual

Acabamos de ver que entrenar una red neuronal es equivalente al problema de optimización de minimizar la función de coste  $C(w, b)$ . Esta función es muy compleja y depende de la arquitectura de la red y de los parámetros  $w$  y  $b$  que conforman esa red. Sin entrar de momento en esta complejidad, vamos a tratar el problema general de cómo minimizar una función, es decir, de **buscar los valores de las variables que proporcionan el valor mínimo**. Esto nos llevará a un algoritmo que podemos usar para entrenar redes neuronales, conocido como *gradient descent*.

### Algoritmo *gradient descent*

Supongamos que queremos minimizar una función  $C(v)$ , donde  $v$  puede ser un vector de varias variables. Una forma de minimizar esta función es recurrir al cálculo y tratar de encontrar el mínimo analíticamente calculando la derivada. Sin embargo, esto no va a ser posible para funciones tan complejas y con tantos parámetros como las que representan las redes neuronales. Para hacer las cosas más fáciles de visualizar, supongamos que  $C$  es una función de dos variables:  $v_1$  y  $v_2$ , como en la figura 5.

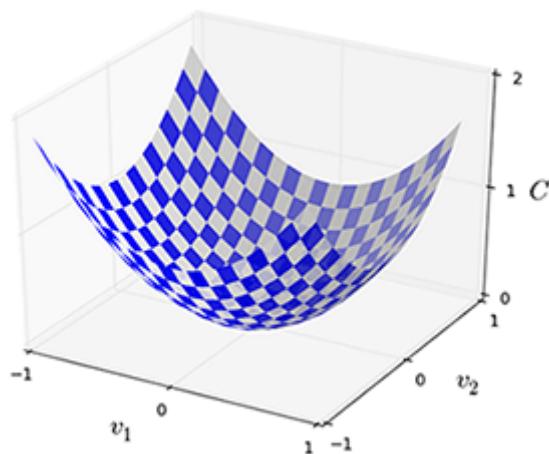


Figura 12. Gráfica simple de una función de dos variables.

Fuente: <http://neuralnetworksanddeeplearning.com/chap1.html>

La idea del algoritmo a desarrollar es la siguiente: empezaremos en un punto  $(v_1, v_2)$  al azar y buscaremos la dirección de máxima pendiente hacia abajo, dando un

pequeño paso en esa dirección. Este proceso se realizará repetidamente hasta llegar al mínimo buscado.

Hagamos esto algo más formal. Según el cálculo, sabemos que la variación en  $C$  con pequeñas variaciones en  $\nu$  sigue la siguiente aproximación:

$$\Delta C \approx \frac{\partial C}{\partial \nu_1} \Delta \nu_1 + \frac{\partial C}{\partial \nu_2} \Delta \nu_2$$

Buscamos hacer  $\Delta C < 0$ , para así minimizar la función. Definimos el vector de variación de  $\nu$  como:

$$\Delta \nu = (\Delta \nu_1, \Delta \nu_2)^T$$

Y el gradiente de  $C$ , que define la dirección de mayor inclinación, como:

$$\nabla C \equiv \left( \frac{\partial C}{\partial \nu_1}, \frac{\partial C}{\partial \nu_2} \right)^T$$

Con esto, podemos reescribir la primera ecuación como:

$$\Delta C \approx \nabla C \cdot \Delta \nu$$

Ecuación 1

Supongamos que elegimos:

$$\Delta \nu = -\eta \nabla C$$

Donde:

- ▶  $\eta$  es una pequeña constante positiva conocida como **learning rate**.

Esto significa que:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$$

Lo cual, como la norma al cuadrado de un vector es siempre un valor positivo, hace que nuestra aproximación sea  $\Delta C \leq 0$ . Por tanto, si elegimos  $\Delta v$  como arriba, estamos haciendo decrecer el valor de la función, tal y como buscábamos.

Con esto, podemos definir la regla de movimiento para movernos de  $v$  a un punto  $v'$  donde  $C$  tenga un valor menor como:

$$v' = v - \eta \nabla C$$

Esta es la **ecuación de movimiento** o ***update rule*** de *gradient descent*. Esta regla será aplicada de manera repetida, de modo que continuaremos calculando el gradiente de  $C$  y actualizando  $v$ , haciendo decrecer  $C$  hasta llegar a un mínimo. Como podemos ver en la figura 6, la idea es calcular el gradiente y dejarnos caer poco a poco por la pendiente que define el valor negativo de este.

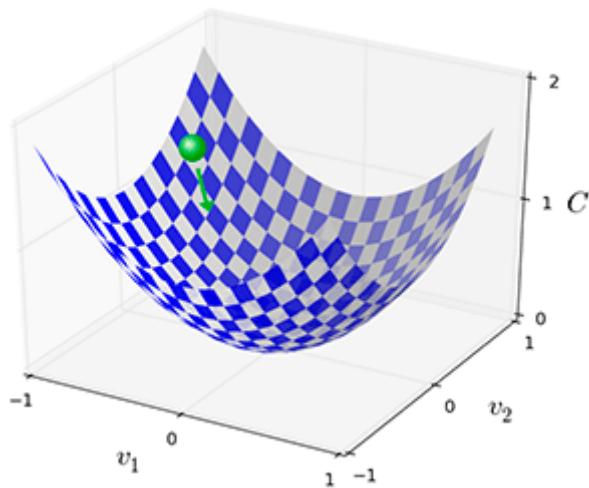


Figura 13. Movimiento de *gradient descent* en busca del mínimo de la función.

Fuente: <http://neuralnetworksanddeeplearning.com/chap1.html>

No hemos comentado el papel de  $\eta$ , el *learning rate*, en este proceso. Como el nombre indica, este valor refleja en cierta manera la **velocidad** a la que *gradient*

*descent* funcionará. La idea es que  $\eta$  sea un valor lo suficientemente pequeño para que tengamos una buena aproximación en la «Ecuación 1».

- ▶ Si elegimos un valor demasiado grande de  $\eta$ , podríamos pasarnos y actualizar  $v$  de modo que incluso vayamos a un valor de  $C$  mayor.
- ▶ Por otro lado, un valor demasiado pequeño de  $\eta$  haría al algoritmo avanzar de manera muy lenta.

La elección de  $\eta$  es por tanto algo complicado, volveremos a tratar el tema en el futuro.

### *Gradient descent* aplicado a redes neuronales

Ahora que conocemos el funcionamiento de *gradient descent* para una función en general, es momento de aplicarlo en nuestro estudio de las redes neuronales. Como sabemos, los parámetros en nuestra función  $C$  son los pesos  $w$  y los *biases*  $b$ , de modo que nuestra posición, en vez de venir dada por un vector  $v$ , viene dada por todos los valores  $w$  y  $b$  de los nodos que componen la red. De este modo, nuestra *update rule* de *gradient descent* para una red neuronal será:

$$w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

Para todos y cada uno de los valores  $w_k$  y  $b_l$ . Sin embargo, notemos un pequeño problema a la hora de entrenar redes neuronales con el algoritmo aquí descrito.

Nuestra **función de coste** tiene la forma:

$$C = \frac{1}{n} \sum_x C_x$$

Donde:

$$C_x = \frac{\|y(x) - a\|^2}{2}$$

Representa el coste de un *training example*. Como vemos, para calcular el gradiente de  $C$ , lo que tenemos que hacer en realidad es calcular el gradiente con respecto a cada punto  $x$  en los datos:

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x$$

¡Esto es demasiado costoso cuando tenemos un gran número de *training examples*! Imaginemos un dataset de un millón de puntos... Si por cada *update* en *gradient descent* necesitamos calcular un millón de gradientes (y cada uno respecto de una gran cantidad de variables), el entrenamiento de redes neuronales se hace casi impracticable.

### *Stochastic Gradient Descent*

Para solucionar este problema, en la práctica las redes neuronales no se entrena calculando el gradiente completo, sino una estimación del mismo obtenida con una muestra aleatoria de *training examples*. Haciendo una media sobre una pequeña muestra de ejemplos, pongamos 128, podemos obtener una aproximación al gradiente real lo suficientemente buena como para minimizar la función sin tener que derivar con respecto a cada punto en el dataset. Se puede pensar en esto como en una especie de encuesta electoral, donde se obtiene una idea del voto sin tener que preguntar a cada uno de los electores. El algoritmo se denomina *stochastic gradient descent*, abreviado comúnmente «para los amigos» como **SGD**.

Más formalmente, los  $m$  *training examples*  $X_1, \dots, X_m$  a elegir se denominan **batch** o **mini-batch**, con  $m$  siendo el tamaño de *batch*. Nuestra estimación es tal que:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^m \nabla C_{X_j}$$

De este modo, nuestra regla de aprendizaje con *stochastic gradient descent* para redes neuronales queda como:

$$w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$
$$b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}$$

Donde:

- ▶  $j$  suma sobre los *training examples* de la *batch*  $X_1, \dots, X_m$ .

El entrenamiento ocurre *batch a batch*. Primero, se elige una muestra aleatoria de  $m$  ejemplos, se calculan los gradientes y se actualizan los parámetros. Este proceso se repite hasta agotar todos los *training examples* del dataset. Cuando esto ha ocurrido, decimos que se ha completado una ***training epoch***. En ese momento, se vuelve a empezar con una nueva *epoch* y así sucesivamente hasta que el entrenamiento esté completo.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 2.4. Backpropagation



Accede al vídeo «*Backpropagation*» a través del aula virtual

Acabamos de ver SGD, un algoritmo utilizado para buscar el mínimo de una función que, aplicado sobre nuestra función de coste, nos lleva a entrenar una red neuronal.

SGD utiliza los gradientes respecto a cada parámetro de la red para actualizar estos mismos parámetros y buscar los valores que llevan al mínimo error o coste. Sin embargo, no hemos visto aún **cómo calcular esos gradientes**.

Una primera idea podría ser, de nuevo, buscar una solución analítica del gradiente de la función final respecto del peso. Pero, tal y como vimos en el apartado anterior, esto se vuelve realmente complejo y difícil para redes neuronales grandes, peor aún si tenemos que buscar una fórmula para cada parámetro individual de la red.

Otra opción sería calcular los gradientes de manera numérica mediante la fórmula de la derivada. Sin embargo, esto se vuelve intratable, ya que necesitaríamos realizar cálculos relativamente complejos por cada uno de los parámetros de la red. Y sabemos que las redes neuronales se caracterizan por poder tener un gran número de ellos.

Sería genial obtener una formulación directa del valor de los gradientes de cada parámetro que fuera computacionalmente eficiente de calcular, permitiendo por tanto entrenar redes neuronales profundas con miles o millones de parámetros. La solución al problema viene dada por el **algoritmo de backpropagation** (o propagación hacia atrás), introducido originalmente en los años 70, pero cuya importancia fue puesta en valor en un famoso trabajo de David Rumelhart y Geoffrey Hinton en 1986, *Learning representations by back-propagation errors*.

Este algoritmo ha sido clave en el desarrollo del *deep learning*, permitiendo la explosión del campo al aportar una forma efectiva de entrenar redes neuronales.

## Expresiones simples de derivadas

Antes de continuar con la exposición del algoritmo de *backpropagation*, veamos un rápido repaso del concepto de derivada. La **fórmula de la derivada** es:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Su significado es la velocidad de cambio de una función con respecto a una variable alrededor de una región infinitesimalmente pequeña a su alrededor.

De la misma manera que podemos hacer derivadas de funciones de una variable, podemos hacer derivadas de funciones de varias variables:

$$f(x, y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

Por ejemplo: si  $x = 4$  e  $y = -3$ , la derivada de  $f$  respecto de  $y$  es 4. Esto significa, intuitivamente, que si incrementamos un poco el valor de  $y$  dejando fijo el resto de variables (en este caso  $x$ ),  $f$  se vería incrementada en 3 veces ese pequeño cambio en  $y$ . Así, podemos ver la derivada como una **medida de la «sensitividad» de una función** respecto a cambios en una de sus variables. Como sabemos, el gradiente en este caso sería:

$$\nabla f = \left[ \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right] = [y, x]$$

Es frecuente utilizar los términos «derivada» y «gradiente» de manera intercambiable y también es común llamar «el gradiente en  $x$ » a la derivada parcial de  $f$  respecto de  $x$ .

Otros ejemplos de valores de derivadas que nos serán útiles son

$$f(x, y) = x + y \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1 \quad \frac{\partial f}{\partial y} = 1$$

$$f(x, y) = \max(x + y) \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1(x \geq y) \quad \frac{\partial f}{\partial y} = 1(y \geq x)$$

## Regla de la cadena

El concepto básico sobre el que se asienta el algoritmo de *backpropagation* es el de la regla de la cadena. Si no estamos muy familiarizados con esta, sería recomendable repasar el concepto con una rápida búsqueda por Internet. Veamos aquí un sencillo ejemplo que nos llevará a nuestro primer ejemplo de *backpropagation*.

Imaginemos la función  $f(x, y, z) = (x + y)z$ . Esta función es muy sencilla y podríamos obtener las derivadas parciales directamente. Sin embargo, para entender el concepto de composición, llamemos  $q = x + y$ , con lo que  $f = qz$ . La regla de la cadena nos dice que:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

De este modo, podemos obtener la derivada parcial de  $f$  respecto de  $x$  mediante el producto de dos derivadas más sencillas. Si no estás muy familiarizado con las derivadas, deberías probar a calcular la derivada de  $f$  respecto de  $x$  tanto directamente a partir de la definición de  $f$  como a partir de la regla de la cadena, y cerciorarte de que el resultado es el mismo.

## Un primer ejemplo de *backpropagation*

Sigamos utilizando nuestra función  $f(x, y, z) = (x + y)z$  para ver un ejemplo básico del algoritmo de *backpropagation*. Imaginemos que ponemos las operaciones de esta función en forma de un circuito, y calculemos paso a paso los valores de  $f$  y de los gradientes respecto a las variables:

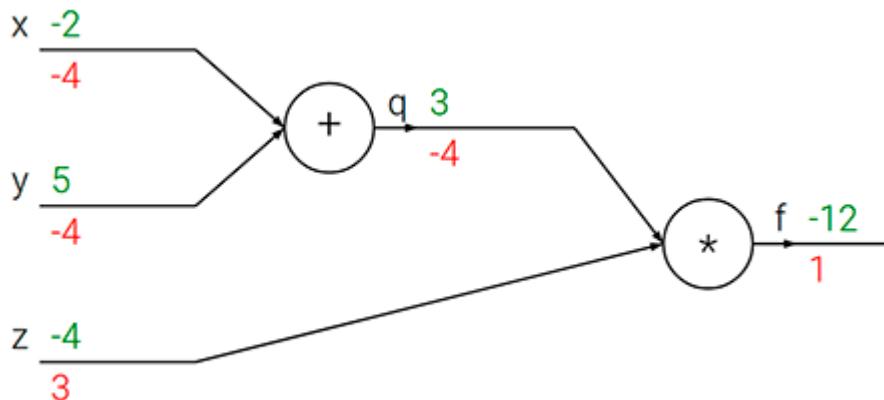


Figura 14.  $f(x, y, z)$  con valores en verde y gradientes en rojo.

Fuente: <http://cs231n.github.io/optimization-2/>

En este ejemplo tenemos como valores iniciales:

$$x = -2$$

$$y = 5$$

$$z = -4$$

De este modo, obtenemos el valor de:

$$q = 3$$

$$f = -12$$

Este primer paso en el que vamos calculando los valores hacia adelante es lo que se conoce como ***forward pass***. Una vez tenemos esto, aplicaremos recursivamente la regla de la cadena hacia atrás en lo que se conoce como ***backward pass***.

Como convención, el primer valor de gradiente sería el gradiente de  $f$  respecto a sí misma, que en este caso es 1, si bien esto no es demasiado importante. Empecemos ahora con  $z$ . Sabemos que  $f = qz$ , y por tanto, la derivada de  $f$  con respecto de  $z$  es  $q$ . Como sabemos del ***forward pass*** que el valor de  $q$  es 3, tenemos que el gradiente en  $z$  es entonces 3. De manera similar, tenemos que el gradiente en  $q$  es  $-4$ .

Necesitamos calcular ahora el gradiente de  $f$  respecto de  $x$ . Por la regla de la cadena, sabemos que:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

- ▶ El gradiente de  $f$  respecto de  $q$  lo acabamos de obtener en el paso anterior y tiene valor  $-4$ .
- ▶ Asimismo, el gradiente de  $q$  respecto de  $x$  es  $1$ , por tanto, al multiplicar ambos valores tenemos que el gradiente en  $x$  es  $-4$ .
- ▶ De manera similar, tenemos el mismo valor para el gradiente en  $y$ .

Podemos ver cómo **el proceso de *backpropagation* es local**. Basta con saber el gradiente local de cada nodo en este circuito con respecto a sus *inputs* y el valor del gradiente que viene propagándose desde el *output*, como se ve en la siguiente imagen:

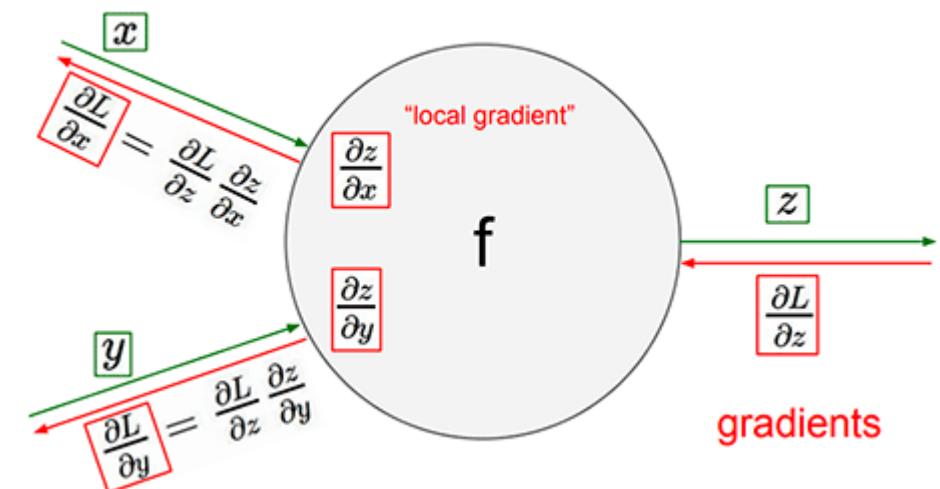


Figura 15. Gradiente local.

Fuente: <http://cs231n.stanford.edu/>

La regla de la cadena nos dice que basta con multiplicar el valor del gradiente que viene propagado desde arriba con el valor del gradiente respecto a cada *input*.

## Un ejemplo ligeramente más complejo

Veamos aquí un ejemplo un poco más complejo aplicado a lo que sería la activación *sigmoid* en una red neuronal. Dejaremos aquí escritos los ingredientes básicos para calcular los valores, lo cual se deja como ejercicio para el alumno.

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

Los valores de las derivadas necesarias son:

$$\begin{aligned} f(x) &= \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2 \\ f_c(x) &= c + x \rightarrow \frac{df}{dx} = 1 \\ f(x) &= e^x \rightarrow \frac{df}{dx} = e^x \\ f_a(x) &= ax \rightarrow \frac{df}{dx} = a \end{aligned}$$

Y el circuito obtenido con sus valores de *forward pass* en verde y de *backward pass* en rojo es:

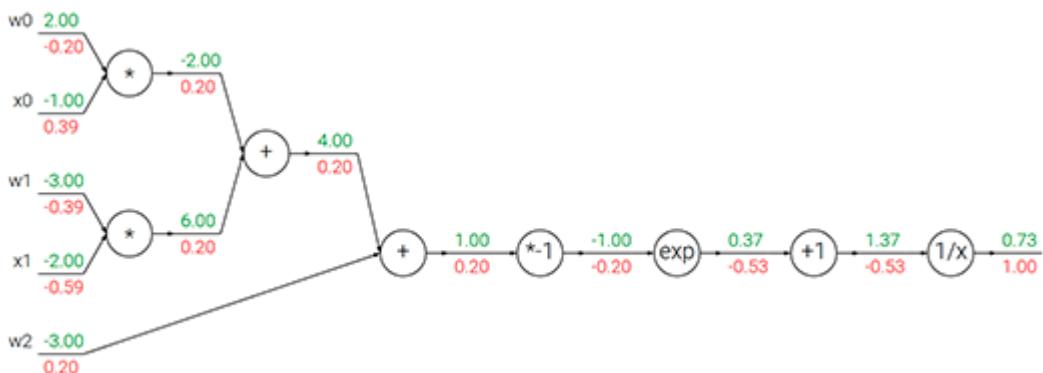


Figura 16. Ejemplo de activación *sigmoid*.  
Fuente: <http://cs231n.github.io/optimization-2/>

## Algunos patrones en *backpropagation*

Al hacer estos ejemplos, es posible que nos hayamos fijado en ciertos patrones que aparecen en las operaciones más comunes que aplicamos.

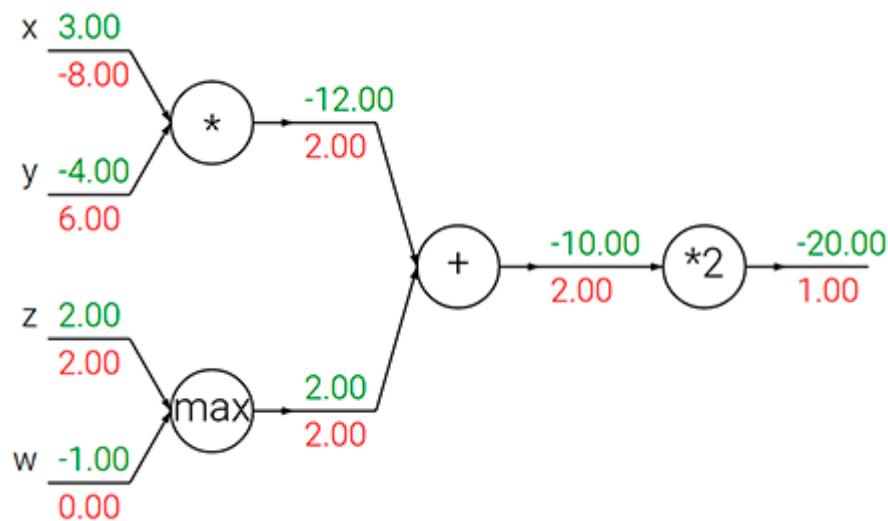


Figura 17. Patrones que aparecen en las operaciones más comunes.

Fuente: <http://cs231n.github.io/optimization-2/>

**Suma:** la suma simplemente coge el gradiente de su salida y lo envía para atrás a todos sus *inputs*.

**Producto:** para un *input* se multiplica el valor del otro *input* por el valor del gradiente de salida.

**Max:** el mayor valor del *input* se lleva todo el valor del gradiente de salida, mientras que el menor *input* se queda gradiente 0.

## Algoritmo de *backpropagation*: recapitulando

Hasta ahora, hemos visto cómo aplicar *backpropagation* a circuitos sencillos de operaciones. Una red neuronal no es más que un circuito o grafo de computación más grande; de hecho, su propia forma es ya la de un grafo de computación donde en cada nodo tenemos una función de activación aplicada al producto de los pesos

por los *inputs* más el *bias*. Si bien podríamos descomponer esto hasta el nivel de operaciones básicas (como hemos hecho en los ejemplos de arriba), también es posible considerar nodos que engloben operaciones más grandes, como la función *sigmoid*.

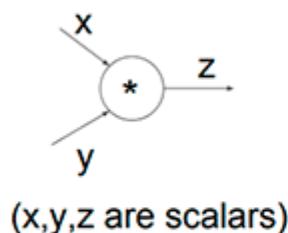
Por ejemplo: la derivada de la función *sigmoid* tiene una derivación analítica muy simple:

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

$$\rightarrow \frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}}\right)\left(\frac{1}{1 + e^{-x}}\right) = (1 - \sigma(x))\sigma(x)$$

En definitiva, para aplicar el algoritmo de *backpropagation*, por cada operación en un nodo que definamos es necesario calcular el valor de la salida en el *forward pass* a partir de sus *inputs* y aportar el cálculo de gradientes de las *inputs* a partir del gradiente propagado.

Por ejemplo, el código para una operación producto sería:



```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Local gradient      Upstream gradient variable

Figura 18. Código para una operación producto.

Fuente: <http://cs231n.stanford.edu/>

La belleza de romper la computación en pequeños nodos radica en que podemos aplicar la misma idea a funciones arbitrariamente complejas. En particular, las arquitecturas de redes neuronales que veremos durante el curso, que usan

conceptos más complejos que las de una simple red neuronal clásica, pueden ser entrenadas igualmente mediante *backpropagation* siempre que sus operaciones sean diferenciables y puedan representarse en forma de un grafo de computación. Como veremos en el tema de frameworks de *deep learning*, así es como funcionan librerías de software como TensorFlow.

Recapitulando, para un solo *training example*  $x$ , el algoritmo final de *backpropagation* quedaría así:

1. *Forward pass*: aplicar el valor de  $x$  a la red neuronal y guardar todos los valores intermedios de salida de cada operación o neurona.
2. *Backward pass*: una vez obtenido el valor final de salida, calcular el valor de la función de coste. La función de coste, que tiene que ser una función diferenciable, también tiene un *input* y unos gradientes respecto a ese *input*. De manera recursiva, propagar el gradiente hacia atrás mediante el uso de la regla de la cadena y calcular los gradientes de cada parámetro.
3. Al terminar el *backward pass*, tendremos calculados los valores de los gradientes de cada parámetro de la red neuronal. Con esto, es posible aplicar *stochastic gradient descent*.

En la práctica (como hemos visto antes), el proceso de *stochastic gradient descent* se hace sobre *mini-batches*. Por tanto, por cada *batch* conteniendo  $m$  *training examples*, calcularemos los gradientes de los parámetros para cada *training example* y, finalmente, aplicaremos la *update rule* sumando sobre los valores de cada *input* en la *batch*:

$$w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$
$$b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}$$

## Varios comentarios finales sobre *backpropagation*

Como hemos visto, el algoritmo de *backpropagation* define, a partir de conceptos matemáticos relativamente sencillos como la regla de la cadena, una forma efectiva de entrenar redes neuronales. Mediante los procesos de *forward* y *backward pass* podemos obtener los valores necesarios para entrenar una red neuronal de complejidad arbitraria sobre una *batch* de *inputs*, evitando el cálculo de complejas derivadas de manera analítica o de gradientes obtenidos numéricos en cada parámetro.

Es importante mencionar que en la práctica se suelen utilizar cálculos vectorizados, donde los parámetros e *inputs* son matrices o vectores. Si bien los gradientes de cada nodo son un poco más difíciles de obtener analíticamente mediante cálculo vectorial, lo cierto es que se siguen las mismas reglas básicas.

Similarmente, en la literatura es frecuente ver el algoritmo de *backpropagation* totalmente desarrollado para el caso de las redes neuronales básicas. La idea es la misma que la vista aquí, pero con todas las cuentas hechas para el caso particular de una red neuronal. Este proceso es un muy buen ejercicio para entender el algoritmo, aunque resulta también algo lioso.



Accede a los ejercicios de autoevaluación a través del aula virtual



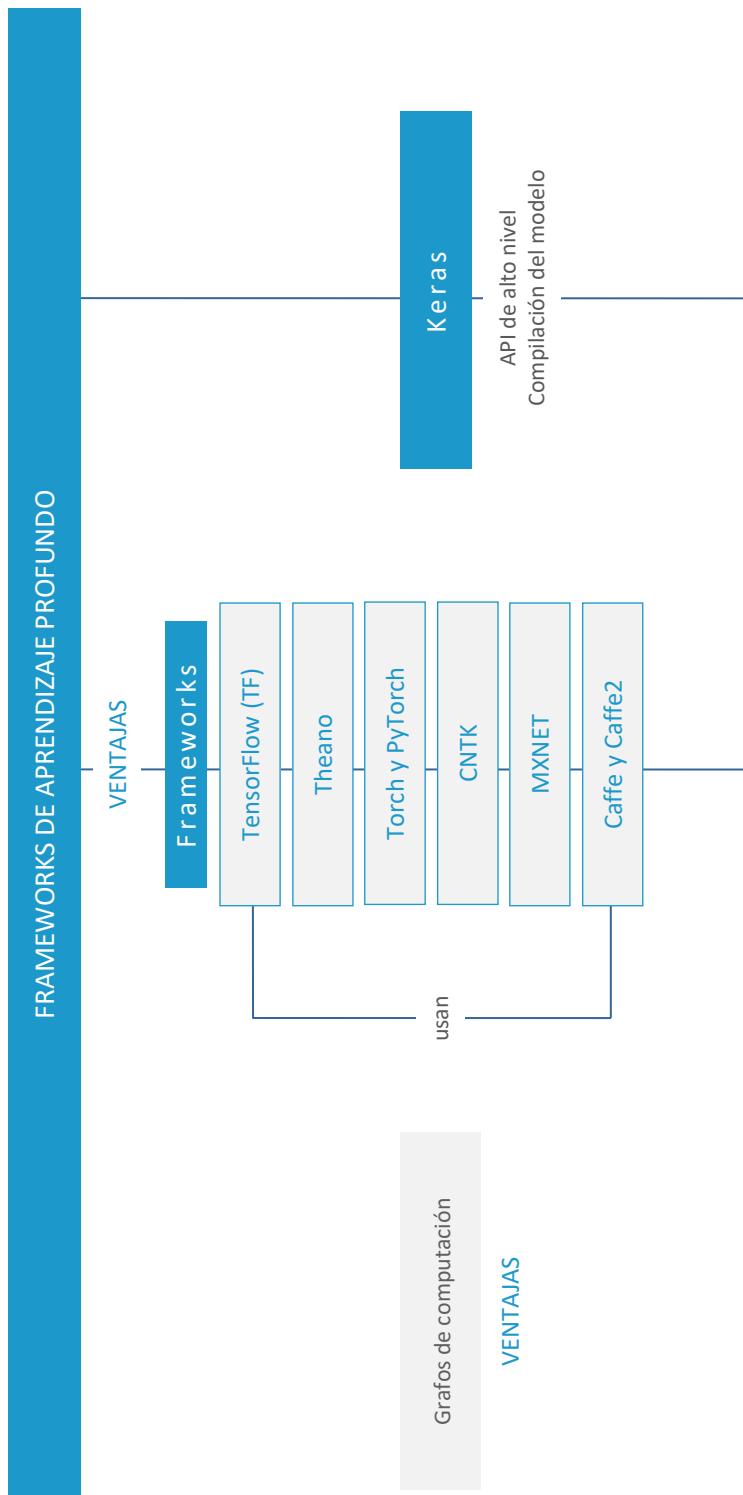
Accede al vídeo «Resumen» a través del aula virtual

# Tema 3. *Frameworks* de aprendizaje profundo

## Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
3.1. Introducción y objetivos	4
3.2. Frameworks de aprendizaje profundo	5
3.3. TensorFlow. Grafos de computación	7
3.4. Otros frameworks	14
3.5. Keras	17
3.6. Referencias bibliográficas	20

# Esquema



## 3.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

En este tema estudiaremos los frameworks de *deep learning* y qué ventajas aportan al entrenamiento de redes neuronales. Profundizaremos un poco más en TensorFlow (el framework más conocido) y en el concepto de tensor y grafo de computación. Asimismo, veremos otros framework utilizados en la actualidad y sus puntos fuertes. Finalmente, echaremos un primer vistazo a Keras, una librería de alto nivel que facilita aún más el entrenamiento de redes neuronales con Python. En este tema:

- ▶ Es necesario comprender el concepto de qué es un framework de aprendizaje profundo y por qué son importantes, relacionando lo visto en este tema con los conceptos básicos de entrenamiento de redes neuronales estudiados durante el curso.
- ▶ También es esencial comprender el concepto de grafo de computación y cómo TensorFlow lo utiliza como base del entrenamiento de modelos. Si bien no es necesario conocer con detalle su funcionamiento interno, de gran complejidad, sí que hay que tener una idea global de cómo se utiliza para entrenar redes neuronales y ser capaces de desarrollar modelos sencillos.
- ▶ El tema sirve igualmente como introducción a Keras, el cual usaremos durante la asignatura. Tenemos que entender su funcionamiento y moverse de manera ágil por su documentación.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 3.2. Frameworks de aprendizaje profundo



Accede al vídeo «Frameworks de aprendizaje profundo» a través del aula virtual

Una de las consecuencias de la reciente popularización del *deep learning* ha sido el surgimiento de una gran cantidad de paquetes de software que facilitan enormemente el entrenamiento de redes neuronales. Como hemos visto, entrenar una red supone la realización de complejas operaciones sobre una gran cantidad de parámetros. Asimismo, implica el cálculo de gradientes que hemos de programar para que el entrenamiento sea correcto.

Si bien esto no es especialmente difícil para redes pequeñas, la cosa se complica con las arquitecturas complejas que iremos viendo durante el curso. El hecho de que haya gran cantidad de decisiones que tomar a la hora de entrenar una red (función de coste, unidad de activación, algoritmo de optimización...), así como operaciones críticas cuyo fallo puede hacer que una red no aprenda, ha llevado a la comunidad de *machine learning* a desarrollar soluciones que permiten facilitar la implementación de modelos de redes neuronales.

**En nuestro caso, entenderemos un framework como una librería o conjunto de librerías de programación que facilita el desarrollo e implementación eficiente de redes neuronales o de algoritmos de *machine learning* en general.**

Presentan grandes **ventajas** en tanto en cuanto:

**Simplifica el desarrollo de grafos de computación de gran tamaño.** La mayoría de frameworks entiende las redes neuronales como un grafo de computación, donde cada nodo en la red es una serie de operaciones a partir de los datos de los nodos anteriores. En vez de tener que codificar las neuronas y sus operaciones nosotros mismos, los frameworks nos dan una serie de abstracciones y operaciones ya

disponibles que podemos conectar en forma de un grafo de computación. Esto abstrae al programador de gran parte de la complejidad en el desarrollo de grandes redes neuronales.

**Calcula de manera automática los gradientes.** Uno de los puntos más complicados en el entrenamiento de redes neuronales, especialmente cuando se utilizan funciones de creciente complejidad, es el de tener que calcular analíticamente y plasmar en código los gradientes necesarios para *backpropagation*. Como este es un proceso en el que es fácil equivocarse, tradicionalmente se hacían comprobaciones numéricas con gradientes aproximados para asegurarse que los cálculos eran correctos y la red neuronal se estaba entrenando correctamente. Los frameworks modernos realizan el *backward pass* de *backpropagation* de manera automática, infiriendo los gradientes correctos a partir del grafo de computación. De esta manera, el programador solo necesita definir la arquitectura de la red para obtener todos los gradientes de manera gratuita.

**Facilita la ejecución y entrenamiento en GPU y entornos distribuidos.** Como veremos más adelante durante el curso, las tarjetas gráficas (GPU) se utilizan con frecuencia para agilizar el entrenamiento de redes neuronales. El uso de frameworks nos permite utilizarlas de manera más sencilla, sin tener que programar código específico para correr en GPU. Similarmente, algunos frameworks como TensorFlow permiten simplificar la ejecución en sistemas distribuidos, donde es posible entrenar utilizando clusters de muchas máquinas.

**Optimiza el entrenamiento y ejecución de los algoritmos.** Del mismo modo, los frameworks suelen utilizar código altamente optimizado, en muchas ocasiones utilizando librerías numéricas que aprovechan al máximo las instrucciones especiales del procesador y sus capacidades multihilo.



Accede a los ejercicios de autoevaluación a través del aula virtual

### 3.3. TensorFlow. Grafos de computación



Accede al vídeo «TensorFlow. Grafos de computación» a través del aula virtual

De todos los frameworks de *deep learning*, probablemente el más utilizado sea TensorFlow (TF): es *open-source* para computación numérica que utiliza grafos de computación donde los datos, en forma de tensores (*Tensor*), «fluyen» (*flow*) entre distintas operaciones. En principio, TF puede utilizarse para una gran cantidad de aplicaciones que precisen de cálculos numéricos, si bien sus aplicaciones comunes giran en torno al *machine learning* y, más en particular, al aprendizaje profundo.



Figura 19. Logo de TensorFlow.

Fue desarrollado por Google, donde es utilizado tanto en investigación como para sistemas en producción. Su código fue liberado como *open-source* en 2015, y desde entonces se ha convertido en un ecosistema software muy completo, con una gran comunidad de usuarios.

Se utiliza principalmente con una API en Python, tal y como haremos en este curso, aunque tiene también API en otros lenguajes como C++ o Java. Sin embargo, la mayor

parte de las operaciones y del núcleo de TF están programados en C++ para una mayor velocidad de ejecución.

## Tensores y grafos de computación

Un tensor, la abstracción básica en TF, es básicamente un array n-dimensional. Por ejemplo:

- ▶ Si tenemos un array de una dimensión, hablamos de un **vector**.
- ▶ Si es de dos dimensiones, hablamos de una **matriz**.
- ▶ Igualmente, un tensor 0-dimensional no es más que un número, un **escalar** en términos algebraicos.

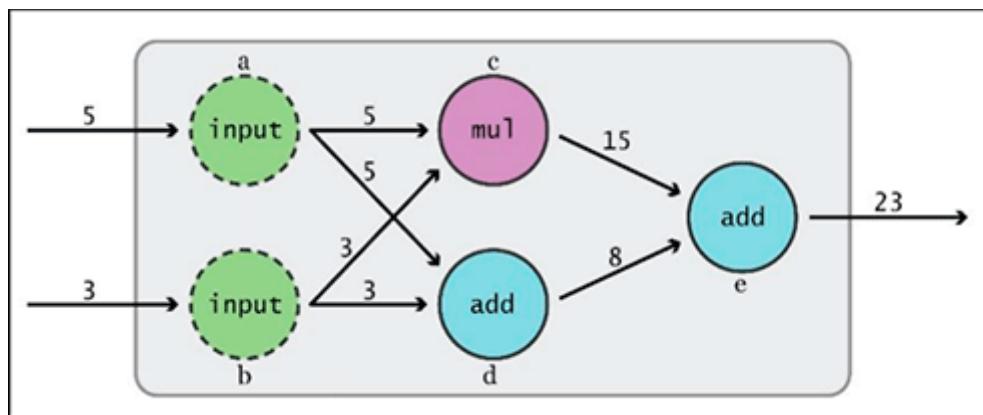


Figura 20. Ejemplo de grafo de computación.  
Fuente: Abrahams, Hafner, Erwitt, y Scarpinelli, 2016.

Un **grafo de computación** es una representación de una serie de operaciones matemáticas. En particular, la representación consiste en un grafo dirigido donde los nodos se corresponden con operaciones y las aristas con datos (en nuestro caso, tensores). En la figura 3 se muestra un ejemplo de un grafo de computación para la función  $f(x,y) = xy + (x + y)$ .

Podemos ver cómo la entrada de la función se representa con dos nodos *input*, los cuales se conectan con dos operaciones: por un lado la multiplicación de  $x$  e  $y$ , y por

otro lado su suma. Las salidas de estas dos operaciones son usadas para realizar una suma final.

Básicamente, TensorFlow funciona mediante la construcción de un grafo de computación y el uso de una sesión que ejecuta las operaciones en el grafo. Con la definición y construcción del grafo, el framework obtiene la información necesaria sobre qué operaciones han de ejecutarse, en qué orden y, además, se asegura de su correcta definición y compatibilidad. Una vez el grafo se ha construido, basta con ejecutar las operaciones con los datos necesarios.

Para aclarar esto, veamos un ejemplo de cómo construir y ejecutar un grafo de computación sencillo con TensorFlow:

```
In [7]: import tensorflow as tf
.....
.... x = 2
.... y = 3
.... op1 = tf.add(x, y)
.... op2 = tf.multiply(x, y)
.... op3 = tf.pow(op2, op1)
.... with tf.Session() as sess:
....     result = sess.run(op3)
.... print(result)
```

Figura 21. Grafo de computación básico en TF.

En este código puede verse cómo definimos dos variables y construimos un grafo mediante la utilización de primitivas de TF. Por ejemplo:

`tf.add()` define la suma de dos elementos.

`tf.multiply` define la multiplicación.

Podemos percibir también que, mediante la utilización de funciones de TF en vez de los operadores matemáticos de Python, estamos definiendo implícitamente la construcción de un grafo de computación; es posible ver el generado con herramientas de visualización como TensorBoard:

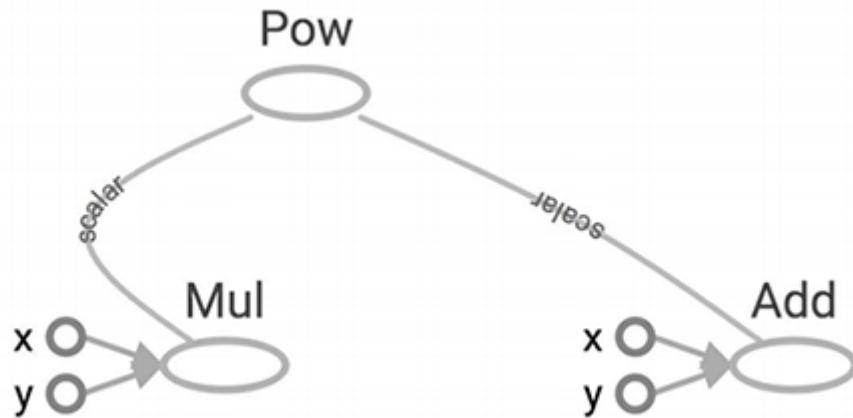


Figura 22. Visualización del grafo de computación del ejemplo anterior mediante TensorBoard.

Finalmente, la sesión instanciada mediante `tf.Session()` es el entorno donde se ejecutan las operaciones definidas en el grafo computacional, lo cual ocurre mediante una llamada a `run()`. La sesión reserva la memoria necesaria para todas las variables empleadas y se encarga de evaluar los tensores.

Es importante comprender que, en el código anterior, las operaciones no se están evaluando línea por línea; la variable `op1` no contiene el resultado de la suma de 3 y 2. Lo que el código está haciendo es construir el grafo de computación que, una vez evaluado con los valores numéricos particulares usados en este ejemplo (2 y 3), proveerá un resultado. De hecho, si ejecutamos el código podemos ver que la variable `op1` es de tipo `tensorflow.python.framework.ops.Tensor` y su contenido es `<tf.Tensor 'Add:0' shape=() dtype=int32>`. Esto no es más que la información que TF va generando sobre el grafo construido. En particular, nos indica varias cosas:

- ▶ El resultado de `tf.add()` aplicado sobre dos escalares de tipo entero es un tensor (`tf.Tensor`).
- ▶ Este tensor es el resultado de la operación definida como `Add:0`. TF va guardando las operaciones creadas con nombres únicos, así como las relaciones entre ellas. Por ejemplo, el hecho de que `op3` dependa de `op1` y `op2` hace que TF guarde estas relaciones, de manera que las operaciones estén relacionadas en un único grafo.

- ▶ El tensor resultante es un escalar, un número, ya que la *shape* es una tupla vacía. Si el resultado fuera una matriz, por ejemplo, veríamos una *shape* del tipo (3,3), indicando que nos encontramos ante una matriz 3x3.
- ▶ El tensor resultante es de tipo `int32`. Esto implica que el tensor contiene enteros. Otros ejemplos de tipos válidos (definidos en `tf.DType`) son `float32`, `float64`, `bool` o `string`.

La evaluación o el momento en el que hacemos «fluir» los datos reales por el grafo y obtenemos el resultado ocurre, como ya hemos dicho, cuando ejecutamos `run()` sobre el objeto sesión. Al hacer `sess.run(op3)`, estamos evaluando el grafo de computación que acaba en `op3`. En ese momento, TF obtiene del grafo todas las operaciones que es necesario ejecutar (en este caso: `op1`, `op2` y `op3`) y rellena los valores de los tensores de salida mediante una evaluación nodo a nodo.

En el caso de que hubiéramos definido otras operaciones en el grafo que no son necesarias para la evaluación de `op3`, estas operaciones serían ignoradas.

## Ventajas de los grafos de computación

¿Por qué TensorFlow y otros frameworks utilizan grafos de computación? Algunas de las razones por las que es conveniente utilizar grafos de computación son las siguientes:

**Optimización de los cálculos a realizar:** las operaciones que no son realmente necesarias no son ejecutadas. Igualmente, ya que todas las operaciones a realizar son conocidas de antemano, se pueden realizar optimizaciones sobre ellas que repercutan en una mayor velocidad de ejecución.

**La ejecución se divide en trozos**, lo que facilita la autodiferenciación. Cuando necesitamos aplicar *backpropagation*, TF recurre al grafo para ver qué operaciones hay que añadir con las derivadas automáticamente calculadas.

**Facilita enormemente la ejecución distribuida** en varias máquinas o GPU. Podemos dividir el grafo en varios subgrafos y hacer que distintos componentes de nuestro sistema distribuido se encarguen de diferentes partes. Por ejemplo, podemos asignar las operaciones de cómputo más complejas a la GPU mientras la CPU se encarga de leer y preprocesar datos.

### Ejemplo completo en TensorFlow

Como muestra de un ejemplo más completo, aquí tenemos el siguiente código donde, utilizando regresión lineal, se intenta predecir la esperanza de vida a partir del número de hijos por familia.

Si bien en este curso no nos detendremos a explicar muchos de los detalles de TensorFlow que se ven en el ejemplo, es importante tener una idea de cómo se entrena un algoritmo de *machine learning* completo con este framework.

```

import tensorflow as tf

import utils

DATA_FILE = "data/birth_life_2010.txt"

# Step 1: read in data from the .txt file
# data is a numpy array of shape (190, 2), each row is a datapoint
data, n_samples = utils.read_birth_life_data(DATA_FILE)

# Step 2: create placeholders for X (birth rate) and Y (life expectancy)
X = tf.placeholder(tf.float32, name='X')
Y = tf.placeholder(tf.float32, name='Y')

# Step 3: create weight and bias, initialized to 0
w = tf.get_variable('weights', initializer=tf.constant(0.0))
b = tf.get_variable('bias', initializer=tf.constant(0.0))

# Step 4: construct model to predict Y (life expectancy from birth rate)
Y_predicted = w * X + b

# Step 5: use the square error as the loss function
loss = tf.square(Y - Y_predicted, name='loss')

# Step 6: using gradient descent with learning rate of 0.01 to minimize loss
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)

with tf.Session() as sess:
    # Step 7: initialize the necessary variables, in this case, w and b
    sess.run(tf.global_variables_initializer())

    # Step 8: train the model
    for i in range(100): # run 100 epochs
        for x, y in data:
            # Session runs train_op to
            sess.run(optimizer,
                     feed_dict={X: x, Y:y})

    # Step 9: output the values of w and b
    w_out, b_out = sess.run([w, b])

```

Figura 23. Ejemplo completo de entrenamiento de un modelo de regresión lineal.

Fuente: [https://docs.google.com/document/d/1kMGs68rIHWIfBiqIU3j\\_2ZkrNj9RquGTe8tJ7eR1sE/edit](https://docs.google.com/document/d/1kMGs68rIHWIfBiqIU3j_2ZkrNj9RquGTe8tJ7eR1sE/edit)

---

Accede a más código en Github a través del aula virtual o desde la siguiente

dirección:

[https://github.com/chiphuyen/stanford-tensorflow-tutorials/blob/master/examples/03\\_linreg\\_placeholder.py](https://github.com/chiphuyen/stanford-tensorflow-tutorials/blob/master/examples/03_linreg_placeholder.py)

---



Accede a los ejercicios de autoevaluación a través del aula virtual

## 3.4. Otros frameworks



Accede al vídeo «Otros frameworks» a través del aula virtual

La popularidad del *machine learning* y del *deep learning*, en particular, ha hecho que la oferta en frameworks se haya multiplicado en los últimos años. La competencia es fuerte, con nuevos *benchmarks* saliendo cada poco tiempo, teniendo en cuenta lo que cuesta entrenar con cada librería algunas de las arquitecturas más populares en aprendizaje profundo. Aquí comentaremos varios de los más importantes.

### Theano



Figura 24. Logo de Theano.

Theano es una librería en Python para computación numérica. Fue creado en 2007 por Yoshua Bengio y es uno de los primeros frameworks que empezaron a utilizarse en el mundo del aprendizaje profundo para optimizar los entrenamientos. Incluye soporte para utilizar GPU y es similar a TensorFlow en cuanto a que las operaciones se definen en forma de un grafo que se ejecuta posteriormente. De hecho, tanto TensorFlow como muchos de los framework modernos deben mucho a Theano, ya que utilizan y perfeccionan varias de las ideas presentes en este. Su desarrollo fue oficialmente interrumpido a finales de 2017.

### Torch y PyTorch



Figura 25. Logos de Torch y PyTorch.

PyTorch es uno de los frameworks más utilizados en la actualidad. Proviene de Torch, otro de los frameworks «clásicos» de *deep learning* como Theano.

- ▶ Está escrito en Lua mientras que PyTorch (como su nombre indica) en Python, lo cual lo hace **más accesible** a la mayoría de desarrolladores e investigadores.
- ▶ Otra de las diferencias que sin duda hacen a PyTorch una versión más moderna de Torch es la **autodiferenciación**. PyTorch está desarrollado por Facebook.
- ▶ PyTorch utiliza grafos de **computación dinámica** en vez de grafos de computación estática, como hace TensorFlow. Un grafo de computación dinámico es calculado a la vez que el código se ejecuta, frente a TF que requiere de dos pasos: definir el grafo y luego ejecutarlo.

Los **grafos de computación dinámica** presentan dos claras ventajas:

- ▶ El código es más sencillo y más parecido a la programación procedural «de toda la vida», lo que además facilita en gran medida la depuración (*debuggear*) del código.
- ▶ El grafo puede cambiar de manera dinámica durante el entrenamiento. Ciertas arquitecturas tienen tamaños que van cambiando según el dato particular que se está evaluando, lo cual hace que el grafo a ejecutar sea distinto en cada iteración. Los grafos de computación dinámica permiten directamente este caso de uso.

TensorFlow también dispone de una variante de ejecución con grafos de computación dinámica llamada Eager Execution, aunque fue añadida más tarde ante el avance de PyTorch y otros.

## Caffe y Caffe2



Figura 26. Logos de Caffe y Caffe2.

Caffe también pertenece a los frameworks «clásicos» de *deep learning*. Está escrito en C++ y su énfasis está en los sistemas en producción con redes neuronales. A diferencia de gran parte de los otros frameworks, no es necesario escribir código para definir modelos: se definen en ficheros de configuración (\*.PROTOTXT).

El uso de Caffe en la actualidad ha descendido mucho dadas sus dificultades para definir modelos más complejos como RNN y CNN, que requieren de larguísimos ficheros de configuración. Sin embargo, aún existen una gran cantidad de sistemas en funcionamiento utilizando modelos entrenados con este framework.

Caffe2, desarrollado de nuevo por Facebook, es la evolución de Caffe y funciona, de manera similar a TensorFlow, mediante grafos estáticos de computación y una interfaz en Python sobre un core escrito en C++.

Es interesante ver cómo Facebook trabaja en dos frameworks, PyTorch y Caffe2, donde el primero está más enfocado a la investigación y el segundo a la puesta de

sistemas en producción. Esta estrategia es distinta a la seguida por Google con TF, que intenta ser un framework válido para ambos aspectos.

## CNTK y MXNET



Figura 27. Logos de CNTK y MXNET.

Para acabar de dejar claro que el mundo de los framework de *deep learning* es un campo de batalla entre los gigantes de la tecnología, los dos últimos frameworks que mencionamos aquí son Microsoft Cognitive Toolkit (CNTK) y Apache MXNet (principalmente apoyado por Amazon). Ambos están pensados como dos frameworks muy escalables y de gran rendimiento. En particular, Amazon facilita la utilización de MXNet en AWS, su ecosistema en la nube.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 3.5. Keras



Accede al vídeo «Keras» a través del aula virtual

TensorFlow y las otras librerías vistas en este tema son frameworks muy potentes y versátiles, capaces de implementar a bajo nivel toda suerte de redes neuronales innovadoras; por «bajo nivel» entendemos el nivel de operaciones matemáticas y de arquitectura de red. Sin embargo, en muchas ocasiones queremos utilizar una arquitectura estándar que nos gustaría definir rápidamente, evitando en la medida de lo posible tener que reescribir el código de los mismos elementos una y otra vez. Es por esto que casi todos los frameworks aquí mencionados disponen de API de alto

nivel que permiten la definición de redes neuronales de una manera más sencilla y directa.

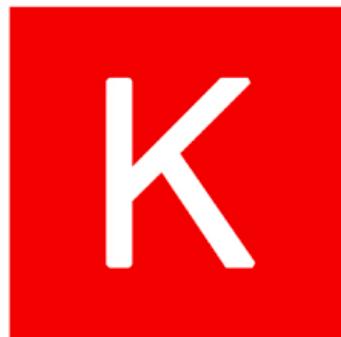


Figura 28. Logo de Keras.

Keras es una de estas librerías de alto nivel, probablemente la primera y más utilizada de ellas. Keras especifica una interfaz modular y *user-friendly* en Python para el desarrollo de redes neuronales, facilitando la tarea a gran parte de los desarrolladores que no necesitan definir arquitecturas de bajo nivel. Internamente, Keras funciona sobre TensorFlow, aunque también es posible usar como *backend* otros frameworks como Theano y CNTK.

La definición de modelos con Keras es realmente sencilla, como podemos ver en el siguiente ejemplo extraído de su página web:

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

# Generate dummy data
import numpy as np
x_train = np.random.randint(1000, 20)
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)
x_test = np.random.randint(100, 20)
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape:
# here, 20-dimensional vectors.
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

Figura 29. Red neuronal con Keras.

Fuente: <https://keras.io/getting-started/sequential-model-guide/>

En el ejemplo podemos ver varios de los elementos estudiados durante el curso. Primero, se generan datos aleatorios. Después, se define el modelo como un modelo secuencial mediante `model = Sequential()`.

Keras dispone de dos estilos de API: **secuencial**, más sencillo y directo, y **funcional**, más versátil, permitiendo arquitecturas más complicadas.

Acto seguido, se van añadiendo al modelo las distintas capas. En este caso, se añaden tres capas Dense (*fully-connected layers*), las dos primeras de 64 unidades y con ‘relu’ como unidad de activación.

La última capa tiene 10 unidades con una activación softmax, lo que indica que es la capa final y que nos encontramos ante un problema de clasificación con 10 clases. Como medida de regularización, entre medias de las distintas capas se añaden unidades de Dropout.

Una vez que la arquitectura de la red está definida, hay que compilar el modelo. Para ello, es necesario especificar una *loss function* y un optimizador. En este caso, debido a que se trata de un problema de clasificación con softmax, la *loss function* es `categorical_crossentropy`, mientras que el algoritmo de optimización es nuestro buen amigo *stochastic gradient descent*, en este caso aderezado con `learning rate decay` y `Nesterov momentum`.

La **compilación del modelo** permite a Keras comprobar que las dimensiones especificadas tienen sentido y que los distintos elementos del modelo son compatibles entre sí. Finalmente, para entrenar, hacemos una llamada a `fit()`.

**Página oficial de Keras:** La página oficial de Keras contiene toda la documentación necesaria para utilizar esta librería así como una gran lista de ejemplos. Imprescindible leer el *Getting started*.

Puedes verlo en el siguiente enlace:

<https://keras.io/>



Accede a los ejercicios de autoevaluación a través del aula virtual

---

### 3.6. Referencias bibliográficas

Abrahams, S., Hafner, D., Erwitt, E. y Scarpinelli, A. (2016). *TensorFlow for Machine Intelligence: A hands-on introduction to learning algorithms*. [Lugar desconocido]: Bleeding Edge Press.



Accede al vídeo «Resumen» a través del aula virtual

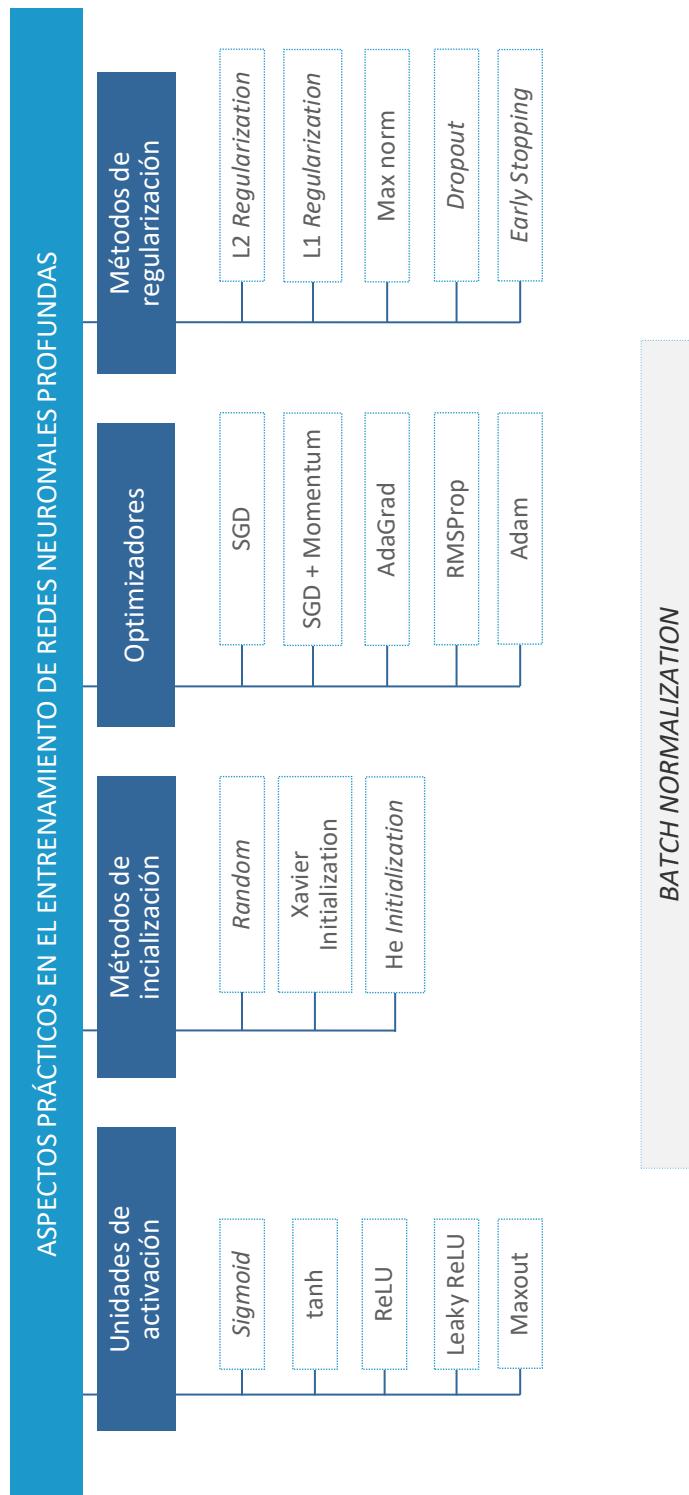
---

# Tema 4. Aspectos prácticos en el entrenamiento de redes neuronales profundas

## Índice

Esquema	3
Ideas clave	4
4.1. Introducción y objetivos	4
4.2. Unidades de activación	5
4.3. Inicialización de parámetros	10
4.4. <i>Batch normalization</i>	13
4.5. Optimización avanzada	16
4.6. Regularización	25
4.7. Referencias bibliográficas	30

# Esquema



## 4.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

Hasta ahora hemos estudiado los conceptos matemáticos básicos que rigen el funcionamiento de las redes neuronales. Como sabemos, una red neuronal es un problema complejo de optimización con muchos parámetros en el que obtener una solución adecuada no es sencillo. En este tema veremos varios de los problemas que aparecen al entrenar redes neuronales profundas y estudiaremos técnicas para entrenarlas de manera más efectiva y eficiente.

Este tema nos ayudará a verificar si hemos comprendido los conceptos básicos vistos en los temas anteriores. Es importante comprender:

- ▶ Las trabas al aprendizaje que aparecen en ciertas unidades de activación, en las formas de inicializar los parámetros de la red y en el algoritmo básico SGD.
- ▶ Cómo otras alternativas facilitan el entrenamiento de las redes neuronales profundas.
- ▶ Qué es el concepto de *overfitting* y cómo las técnicas de regularización ayudan a combatirlo.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 4.2. Unidades de activación



Accede al vídeo «Unidades de activación» a través del aula virtual

Uno de los elementos más críticos en las redes neuronales son las unidades de activación o *non-linearities* presentes en las neuronas de la red. Hemos visto ya el funcionamiento de la unidad *sigmoid*, utilizada de manera clásica en el mundo de las redes neuronales. Uno de los avances en el mundo del *deep learning* que ha permitido el entrenamiento de redes más profundas y complejas ha sido la irrupción de nuevas unidades de activación que solventan varios de los problemas que esta unidad presenta.

### Sigmoid

Como ya sabemos, la función *sigmoid* se define de la siguiente manera:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

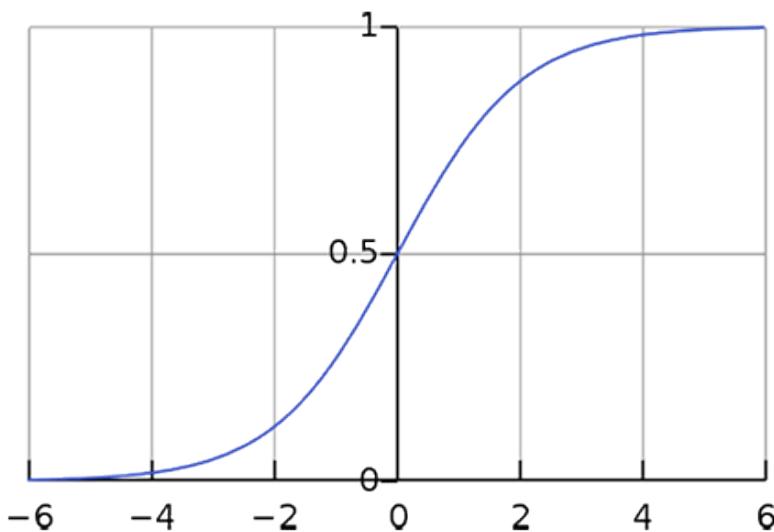


Figura 30. Gráfica de la función *sigmoid*.

Fuente: <https://github.com/stanfordnlp/cs224n-winter17-notes/blob/master/notes1/fig/sigmoid.png>

La unidad se caracteriza por convertir cualquier número real en un número entre 0 y 1, haciendo que valores positivos grandes sean prácticamente 1, mientras que valores negativos grandes tienden a 0.

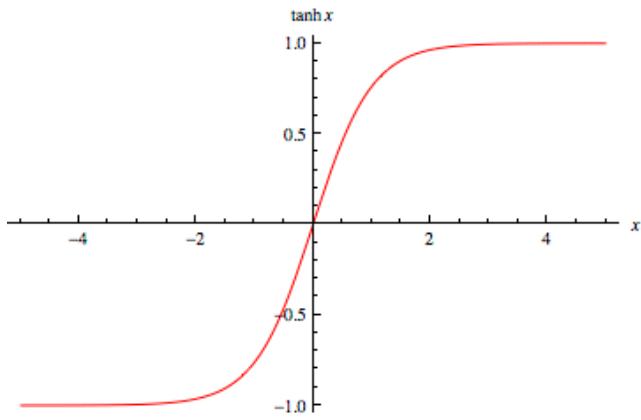
Las unidades *sigmoid* han caído bastante en desuso en las arquitecturas modernas debido a una serie de **desventajas**:

**Las unidades *sigmoid* saturadas «matan» los gradientes.** Como podemos apreciar en la gráfica, para valores altos positivos o negativos, la unidad queda saturada en valores cercanos a 1 y -1, donde la pendiente de la gráfica es nula. En otras palabras, la derivada o gradiente de la función es 0. Como sabemos, durante la propagación hacia atrás, el gradiente local de la unidad es multiplicado por el gradiente de la salida de la unidad. Esto implica que, si el gradiente de la función *sigmoid* es casi 0, el valor del producto también lo será, «matando» de manera efectiva el gradiente que se propaga y por tanto eliminando la señal e impidiendo el aprendizaje.

**La salida no está centrada en 0.** Los valores de *sigmoid* son estrictamente positivos, lo cual crea ciertas dinámicas no deseables en el entrenamiento de las redes. En particular, si una neurona recibe todos sus *inputs* con valores positivos, la derivada respecto de los pesos *w* será siempre positiva o negativa, creando una ineficiencia en el aprendizaje. El hecho de que en este caso la salida no esté centrada en 0 provoca estas dinámicas en las siguientes capas de la red.

**Implica el cálculo de una función exponencial  $e^x$ .** Si bien esto no es un cuello de botella a la hora de hacer cálculos, el cálculo de una exponencial conlleva cierta complejidad en comparación con otras operaciones.

## Tanh



Fuente: <http://mathworld.wolfram.com/HyperbolicTangent.html>

La **tangente hiperbólica**,  $\tanh(x)$ , presenta una forma bastante similar a *sigmoid*: de nuevo saturando por los lados, pero esta vez entre los valores -1 y 1. La gran ventaja respecto a *sigmoid* es que la salida está centrada alrededor de 0. Sin embargo, debido a la saturación de valores, la unidad sigue teniendo el problema de «matar» a los gradientes. En la práctica, es siempre preferible usar  $\tanh$  antes que *sigmoid*.

## ReLU

La unidad ReLU (*rectified linear unit*) tiene la siguiente fórmula:

$$f(x) = x^+ = \max(0, x)$$

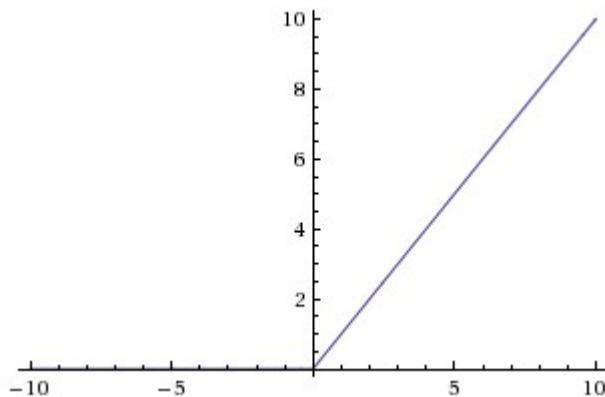


Figura 32. Gráfica de la unidad ReLU.

Fuente: <https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning>

La unidad ReLU tiene salida 0 para valores menores que 0 y la función identidad para valores mayores que 0. Es la unidad de activación más usada en la práctica.

Sus **ventajas** son:

- ▶ No satura en el régimen positivo. La tendencia a «matar» gradientes que veíamos en *sigmoid* y *tanh* desaparece para valores mayores que 0.
- ▶ Computacionalmente muy eficiente. La unidad ReLU no requiere del cálculo de exponenciales. Es suficiente con aplicar un *threshold* y convertir todos los números negativos en 0. El resto de valores se quedan igual.
- ▶ Acelera la convergencia de *Stochastic Gradient Descent*. Hasta seis veces más rápida que las unidades *sigmoid* o *tanh*, según Krizhevsky et al. (2012).

Por otro lado, las ReLU también tienen algunos **problemas**:

- ▶ Como pasaba con *sigmoid*, la salida no está centrada en 0.
- ▶ Las ReLU son una unidad frágil y pueden «morir» durante el entrenamiento. Es posible que un cambio grande en el gradiente haga que sus *weights* cambien de manera que la unidad no consiga volver a activarse nunca, es decir, no producir nunca más un valor mayor que 0. Esto provoca que el gradiente que fluye por la unidad sea siempre 0 a partir de ese momento, efectivamente convirtiendo a la neurona con la ReLU en una neurona «muerta». De hecho, es normal que al entrenar redes neuronales con ReLU veamos que un gran número de ellas nunca se activan ante ningún *input* del *training set*, si bien esto no tiene por qué desembocar en un mal rendimiento de la red. Normalmente, el problema de las unidades muertas mejora con *learning rates* pequeños.

### Leaky ReLU

La Leaky ReLU es una variante de la ReLU que intenta solucionar el problema de las ReLU «muertas». Básicamente, la función añade una pequeña pendiente en el régimen negativo, evitando que la unidad pueda morir:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

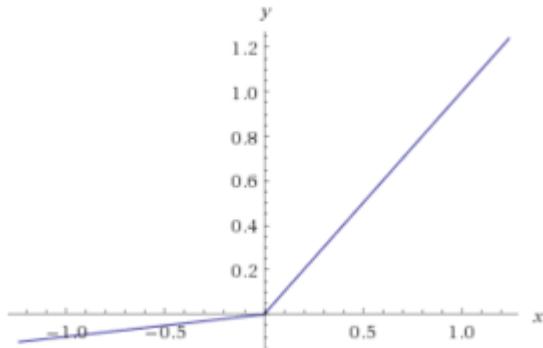


Figura 33. Gráfica de la unidad ReLU.

Fuente: <https://datascience.stackexchange.com/questions/5706/what-is-the-dying-relu-problem-in-neural-networks>

Una variante de la Leaky ReLU es la PReLU, la cual añade un parámetro (que también es aprendido) en vez de 0.01 como pendiente, permitiendo mayor flexibilidad.

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases}$$

Finalmente, otra variante es la ELU, que intenta resolver el problema de las activaciones no centradas en 0, lo cual hace que el entrenamiento sea más lento.

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ a(e^x - 1) & \text{otherwise} \end{cases}$$

## Maxout

Maxout es otra unidad propuesta para solucionar el problema de las ReLU que mueren durante el entrenamiento. La unidad generaliza tanto a la ReLU como a la Leaky ReLU y presenta una forma un tanto distinta de las clásicas unidades que hemos visto de la forma  $f(w \cdot x + b)$ . Su fórmula es:

$$\max (w_1^T x + b_1, w_2^T x + b_2)$$

Como vemos, la unidad introduce dos parámetros  $w$  en vez de uno. Esta unidad se convertiría en ReLU con  $w_1 = 0$  y con  $b_1 = 0$ . Si bien es más versátil y no «muere» como las ReLU, esta unidad tiene el problema de que necesita el doble de parámetros.

### ¿Cuál usar?

Tenemos un gran surtido de unidades de activación para elegir. En general, en la actualidad se recomienda utilizar ReLU, teniendo cuidado con la *learning rate* y monitorizando el entrenamiento para que no haya demasiadas unidades muertas. Si esto es un problema se puede probar alguna variante de Leaky ReLU o Maxout. En general, debería evitarse el uso de *sigmoid* por la gran cantidad de problemas que presenta.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 4.3. Inicialización de parámetros



Accede al vídeo «Inicialización de parámetros» a través del aula virtual

Hemos visto que las redes neuronales tienen un gran número de parámetros que aprender: en particular, los pesos  $w$  y los biases  $b$ . También hemos repasado cómo el proceso de aprendizaje se realiza mediante *stochastic gradient descent*. Sin embargo, ¿con qué valores inicializamos todos estos parámetros? Esta pregunta es muy importante, ya que los valores iniciales de los parámetros tienen una gran repercusión en el proceso de entrenar una red neuronal.

### Error: inicializar todo a 0

Podríamos pensar que una forma de inicializar nuestros parámetros es en un punto medio, por ejemplo con todo 0. Sin embargo, esto sería un grave error. Si todas las neuronas de la red tienen la misma salida (lo cual pasaría si inicializamos todo a 0), todas ellas tendrán el mismo gradiente durante la *backpropagation* y, por tanto, cambiarán los parámetros de igual manera. Es necesaria cierta asimetría en los valores de inicialización para evitar este fenómeno.

## Mejor: inicialización aleatoria

Para evitar el problema de la inicialización a todo 0, necesitamos «romper la simetría» en la red. Una opción sencilla es inicializar los pesos de manera aleatoria usando una distribución normal centrada en 0 y con un valor pequeño de desviación típica. Por ejemplo:

```
W = 0.01* np.random.randn(fan_in, fan_out)
```

Al hacer todos los pesos aleatorios y por tanto distintos, conseguimos que los *updates* del gradiente sean distintos y, así, la red consiga aprender. Es importante destacar que los números tienen que ser pequeños (de aquí el valor de 0.01), ya que unos pesos grandes podrían hacer que las funciones de activación se saturasen frecuentemente.

Si bien esta solución es correcta, aún tiene ciertos problemas. En redes muy profundas, el hecho de que los pesos se inicialicen con números pequeños puede dar problemas a la hora de entrenar, haciendo que los valores en la red se vuelvan más y más pequeños y que los gradientes tiendan a 0.

## Xavier initialization y He initialization

El caso de arriba es un ejemplo claro de la dificultad de trabajar con redes neuronales profundas, donde cualquier problema a la hora de inicializar los parámetros puede impedir que la red entrene correctamente. La inicialización de parámetros es un

campo de estudio activo, con continuos avances en busca de mejores estrategias de inicialización.

Una solución al problema visto en el apartado anterior es la conocida como **Xavier initialization**. La idea aquí es intentar que la varianza de salida de una neurona sea igual a la varianza de entrada. Esto hace que todas las neuronas de la red tengan aproximadamente la misma distribución de salida, lo que acelera la convergencia al entrenar. En particular, para un peso  $w$  individual una formulación de esta inicialización es:

```
w = np.random.randn(n) * sqrt(2/(n_in + n_out))
```

Donde:

- ▶  $n_{in}$  es el número de *inputs* de la neurona.
- ▶  $n_{out}$  el número de neuronas en la capa siguiente.

Un estudio más reciente (He, Zhang, Ren y Sun, 2015), deriva una inicialización específica para unidades ReLU:

```
w = np.random.randn(n) * sqrt(2/n)
```

Donde:

- ▶  $n$  es el número de *inputs* de la neurona.

En la práctica, se recomienda utilizar esta inicialización cuando entrenamos con redes neuronales con ReLUs.

## Inicialización de los *biases*

Hasta ahora, hemos hablado de inicializar los pesos  $w$ , pero no sobre los biases  $b$  de la red neuronal. Estos no son tan críticos una vez que los pesos  $w$  son inicializados

aleatoriamente, lo que ya rompe la simetría de la red. Es muy común inicializarlos simplemente a 0.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 4.4. Batch normalization



Accede al vídeo «*Batch normalization*» a través del aula virtual

Uno de los mayores avances en el entrenamiento de redes neuronales profundas en los últimos tiempos ha sido la técnica de ***batch normalization***. En gran medida, esta técnica evita muchos de los problemas creados por una mala inicialización de parámetros y posibilita una mejor convergencia al entrenar redes neuronales.

En los puntos anteriores hemos explicado que la distribución de valores de salida en las unidades de activación tiene una gran repercusión en el entrenamiento de una red neuronal. La idea de *batch normalization* consiste en forzar que la entrada de las unidades de activación en todas las capas siga una distribución normal estándar. Para ello, se obtiene la media y la varianza empíricas de los valores por ***training batch***. Esto es, por cada batch que entrenamos, obtenemos una media y las varianzas a partir de todos los *inputs* en esa batch. Esta normalización se hace por cada dimensión del *input* mediante la siguiente fórmula:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

Normalmente, la operación de *batch normalization* se ejecuta **antes de la aplicación de la función de activación** y, por tanto, justo después de la operación lineal  $Wx + b$ , donde  $x$  es el *input* de la neurona.

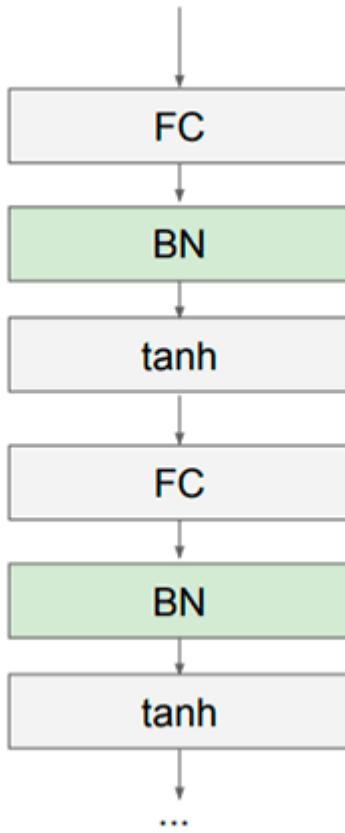


Figura 34. Aplicación de *batch normalization*.

Fuente: <http://forums.fast.ai/t/lesson-2-using-batch-normalization-after-non-linearity-or-before-non-linearity/4817>

Si normalmente tenemos una función de activación  $f(Wx + b)$ , la normalización se ejecuta al calcular  $Wx + b$  y antes de aplicar la *non-linearity*.

Esta normalización capa a capa puede reducir el poder de expresión de una red neuronal. Por tanto, es habitual reemplazar los valores de la normalización por una nueva parametrización con dos nuevos parámetros:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Los nuevos parámetros introducidos son aprendidos durante el entrenamiento como si fueran unos pesos, es decir, también se aplican las técnicas de optimización de *gradient descent* sobre ellos. De este modo, la red tiene la capacidad teórica de aplicar la función identidad y por tanto «deshacer» la *batch normalization* si así quisiera.

En total, el **proceso de batch normalization** queda resumido en la siguiente imagen:

<b>Input:</b> Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$ ;	
Parameters to be learned: $\gamma, \beta$	
<b>Output:</b> $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$	
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$	// mini-batch mean
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$	// mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$	// normalize
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$	// scale and shift

Figura 35. Proceso de *batch normalization*.

Fuente: <https://arxiv.org/pdf/1502.03167.pdf>

Durante la inferencia (esto es, una vez la red neuronal ya ha sido entrenada y la estamos usando para predecir), las medias y varianzas se fijan con valores totales obtenidos durante el entrenamiento.

Las **ventajas** de la *batch normalization* son las siguientes:

- ▶ Mejora el flujo de gradientes por la red durante el aprendizaje, aumentando por tanto la velocidad de convergencia.
- ▶ Permite *learning rates* mayores.
- ▶ Reduce la dependencia en la inicialización de parámetros que tiene el proceso de entrenamiento.
- ▶ Puede interpretarse como una forma de regularización.

Con todas estas, la técnica de *batch normalization* es muy usada en la práctica.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 4.5. Optimización avanzada



Accede al vídeo «Optimización avanzada» a través del aula virtual

En temas anteriores, hemos visto cómo el entrenamiento de una red neuronal se fundamenta en solucionar un problema de optimización. Para ello, estudiamos *stochastic gradient descent*, un algoritmo sencillo que consiste en aproximar localmente de manera lineal la función que intentamos minimizar, y realizar pequeños pasos en la dirección de máximo descenso. En esta sección, veremos algunos problemas de este método de optimización y algunas técnicas más avanzadas para entrenar redes neuronales.

### Problemas de SGD

Fórmula de *update* de SGD:

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

Si bien *stochastic gradient descent* es un algoritmo que funciona correctamente en general, tiene una serie de problemas que dificultan en ocasiones el entrenamiento de redes neuronales. Por ejemplo, supongamos que tenemos una región donde el coste o *loss* disminuye rápidamente en una dirección y muy lentamente en otra, como se ve en la siguiente imagen.

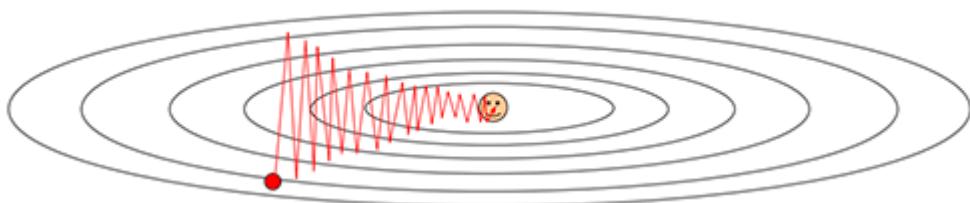


Figura 36. Ejemplo de problema de SGD.

Fuente: <http://cs231n.github.io/>

En la imagen, el punto rojo es el punto de inicio, mientras que el punto central representa el mínimo a alcanzar. Como vemos, SGD empieza un movimiento en zigzag siguiendo la dirección de máximo cambio, encontrando el mínimo de manera muy ineficiente. El camino óptimo habría sido seguir una línea directa de bajada hacia el punto central. Este problema es aún peor en el caso de una red neuronal real, donde en vez de dos dimensiones como en la imagen, podemos llegar a tener millones de variables, dando lugar a millones de posibles direcciones que el gradiente puede tomar.

Otro gran problema es el de los **mínimos locales**: puntos mínimos de una función que no son el mínimo global de esta (figura 8).

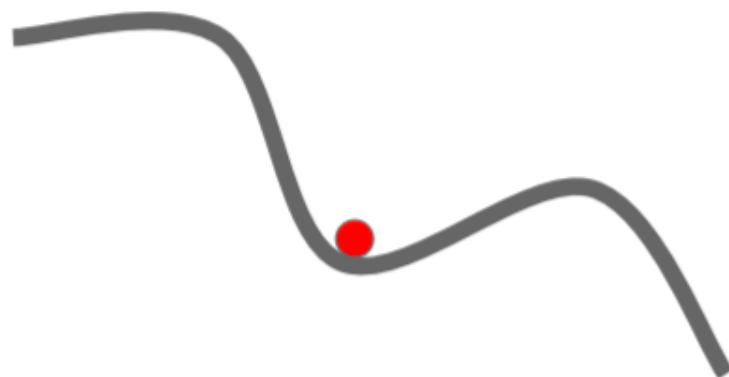


Figura 37. Ejemplo de mínimo local.

Fuente: <http://cs231n.github.io/>

De manera similar, los **puntos de silla (saddle points)** representan el mismo problema. Si bien no tienen la forma de un mínimo, en ellos la derivada también es cero:

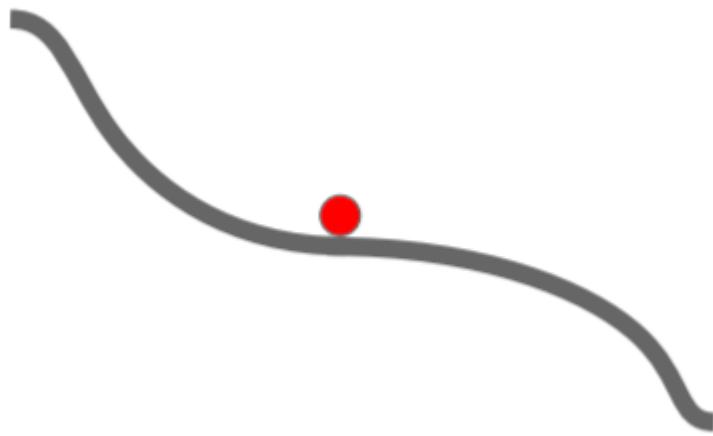


Figura 38. Ejemplo de mínimo local.

Fuente: <http://cs231n.github.io/>

En el caso particular de las redes neuronales, donde tenemos una gran cantidad de parámetros, los puntos de silla aparecen de manera mucho más frecuente que los mínimos locales.

Es importante mencionar aquí que el entrenamiento de redes neuronales es un **problema de optimización no convexo**, para el cual encontrar la solución óptima no está garantizado; el hecho de que tengamos un gran número de parámetros en la red hace más complejo aún encontrar la solución óptima. Hablamos de funciones con, potencialmente, millones de variables a optimizar, donde es prácticamente imposible de visualizar un proceso de minimización.

Lo que sí podemos imaginar es que encontrar el mínimo global de una función de este estilo es computacionalmente intratable y, en muchas ocasiones, tenemos que conformarnos con que el mínimo encontrado sea lo suficientemente bueno. Hay que resaltar aquí que al inicializar los pesos de la red de manera aleatoria, las soluciones encontradas suelen ser distintas unas a otras. En la práctica por suerte, al entrenar una red desde distintos valores iniciales se tiende a encontrar mínimos parecidos, aunque este no tiene por qué ser el caso al ser un proceso dependiente del punto inicial, de la función a tratar (la topología de la red junto con sus parámetros) y del algoritmo de optimización en uso.

Es también importante mencionar que, sin embargo, encontrar un mínimo global o un valor demasiado cercano al mínimo global no es lo que queremos siempre. Las redes neuronales tienen una gran capacidad de representación, por lo que podríamos encontrarnos con que pueden explicar perfectamente los datos de entrenamiento, pero no unos datos fuera del *training set*. Esto es lo que se conoce como ***overfitting***, y lo veremos en el siguiente punto del tema.

## SGD + Momentum

Una variante de SGD para resolver los problemas que hemos visto en el apartado anterior consiste en añadir *momentum*. En SGD con *momentum* tenemos un vector de velocidad que es una especie de media ponderada de gradientes anteriores:

$$\begin{aligned}v_{t+1} &= \rho v_t + \nabla f(x_t) \\x_{t+1} &= x_t - \alpha v_{t+1}\end{aligned}$$

Como vemos en las ecuaciones, la regla de *update* del punto  $x$  depende ahora también de un nuevo término, la **velocidad**. Por su parte, el parámetro  $\rho$  corresponde a la fricción y su valor suele ser 0.9 o 0.99.

La idea es que el descenso va adquiriendo «velocidad» en las direcciones correctas a través de todos los *updates*, acelerando el proceso de convergencia. SGD con *momentum* soluciona, además, los problemas que hemos visto anteriormente.

En el caso de los mínimos locales y los puntos de silla, si bien el gradiente es 0, al tener el vector velocidad información de los gradientes anteriores podemos «pasarnos» estos puntos conflictivos:

## Local Minima      Saddle points



Figura 39. Ejemplo de SGD + Momentum ante mínimo local y punto de silla.

Fuente: <http://cs231n.github.io/>

De la misma manera, los ineficientes zigzags que veíamos antes se compensan unos a otros gracias al vector de velocidad, permitiendo encontrar una solución al problema más efectiva:

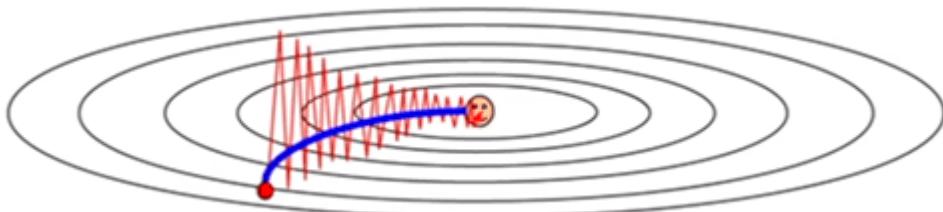


Figura 40. Ejemplo de SGD + Momentum como solución.

Fuente: <http://cs231n.github.io/>

Una variante más moderna de SGD con momentum es el denominado **Nesterov Momentum**. Esta técnica goza de mejores propiedades de convergencia teóricas para funciones convexas y, aunque el entrenamiento de redes neuronales no optimiza dicho tipo de funciones, en la práctica parece que también se comporta mejor. La idea de Nesterov Momentum es aplicar el gradiente sobre el *update* que hace el vector velocidad, puesto que este va a afectar a la posición igualmente.

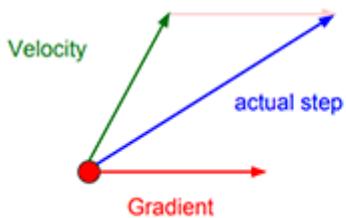
Ecuaciones de Nesterov Momentum:

$$v_{t+1} = p v_t - \alpha \nabla f(x_t + p v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$

A continuación, una comparación entre las *update rules* de SGD con Momentum y con Nesterov Momentum:

Momentum update:



Nesterov Momentum

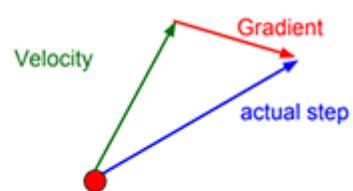


Figura 41. Nesterov *update* y Nesterov Momentum.

Fuente: <http://cs231n.github.io/>

## AdaGrad

AdaGrad es un método que modifica la *learning rate* de manera independiente para cada parámetro. Encontrar una *learning rate* adecuada es una tarea compleja, por lo que el atractivo de este método, junto con varios de los que veremos a continuación, es que tenemos que preocuparnos menos de ello. Las ecuaciones de AdaGrad pueden simplificarse visualmente en código NumPy de la siguiente manera:

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Figura 42. Ecuaciones AdaGrad con NumPy.

Fuente: <http://cs231n.github.io/>

Aquí, las variables  $x$  y  $\text{grad\_squared}$  son vectores, por lo que las operaciones se realizan elemento a elemento. La idea de AdaGrad consiste en dividir el valor del *update* de  $x$  por el cuadrado del gradiente:

- ▶ Si el valor del gradiente es grande, estamos haciendo que el avance en esa coordenada sea reducido.

- ▶ Si es más pequeño, hacemos que el avance sea amplificado.

En nuestra ya conocida imagen, estamos desacelerando el movimiento en la dirección vertical mientras que lo aceleramos en la horizontal

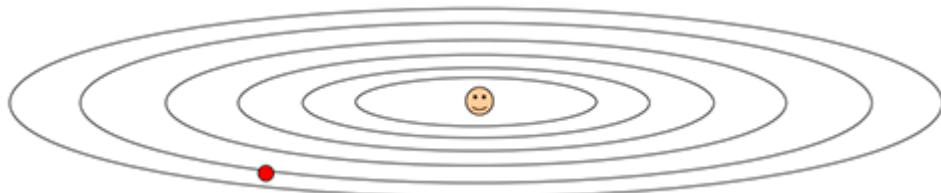


Figura 43. Ejemplo con AdaGrad.

Fuente: <http://cs231n.github.io/>

Es importante fijarse en que `grad_squared` va sumando valores, por lo que a lo largo del entrenamiento los pasos que demos serán cada vez más pequeños. La idea aquí es que, una vez estemos más cerca del mínimo, necesitamos hilar más fino, mientras que al principio podemos avanzar de manera más rápida en busca de las direcciones adecuadas. Pero esto crea un problema para AdaGrad y es que esta reducción de velocidad llega a ser muy agresiva y podría detener el aprendizaje demasiado pronto.

## RMSProp y Adam

RMSProp y Adam son dos **métodos de optimización** que elaboran las ideas de AdaGrad.

**RMSProp**, en vez de sumar los cuadrados de los gradientes, introduce un `decay`, de manera que los valores no se acumulen y evitando que el entrenamiento se pare en cierto momento:

```

grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)

↓

grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)

```

AdaGrad                                    RMSProp

Figura 44. Comparación entre AdaGrad y RMSProp.

Fuente: <http://cs231n.github.io/>

**Adam** es un optimizador que, utilizado en su forma por defecto, presenta muy buenos resultados. Combina RMSProp con la idea de Momentum vista en SGD.

```

first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))

```

Momentum

Bias correction

AdaGrad / RMSProp

Figura 45. Método Adam.

Fuente: <http://cs231n.github.io/>

### Learning rate decay

Todos los algoritmos que hemos visto hasta ahora utilizan *learning rates*, que es probablemente el hiperparámetro más crítico a elegir a la hora de entrenar una red neuronal. En la siguiente imagen podemos ver una gráfica de *loss* por época de entrenamiento. Una buena elección en *learning rate* nos debería llevar a una reducción de la *loss* hasta converger en un valor bajo. Una *learning rate* alta también lleva a convergencia, pero el hecho de que no sea lo suficientemente pequeña hace que converja en una solución no óptima. Si la *learning rate* es demasiado alta, nos encontramos probablemente dando saltos sin sentido en el espacio de parámetros (*overshooting*) y por tanto, la *loss* crece en vez de disminuir. Finalmente, en el caso

de una *learning rate* demasiado baja, la red neuronal se entrena correctamente, pero a un ritmo demasiado lento.

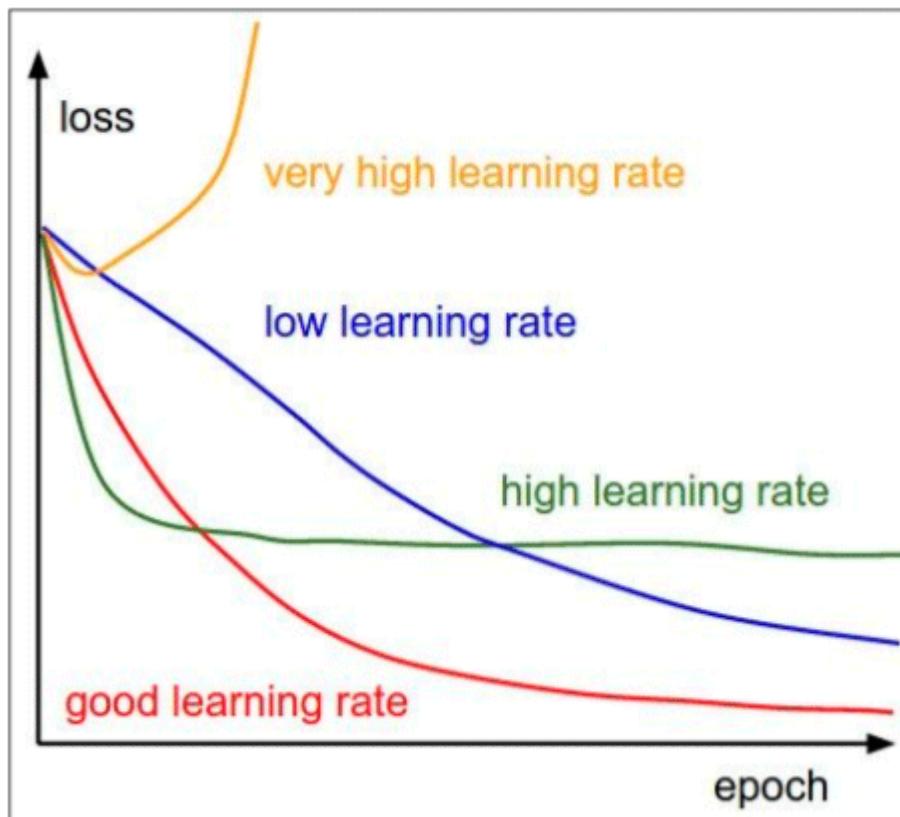


Figura 46. Método Adam.

Fuente: <http://cs231n.github.io/>

Una idea para tratar de una manera más eficiente la *learning rate* consiste en usar *learning rate decay*, haciendo que el *learning rate* se vaya reduciendo a medida que avanza el entrenamiento. De este modo, el entrenamiento comienza con una *learning rate* mayor, haciendo que la *loss* se reduzca rápido, y poco a poco esta se va reduciendo permitiendo buscar mejor el espacio de soluciones.

Hay varias estrategias posibles para ejecutar el *decay*:

- ▶ *Step decay*: reducir, por ejemplo a la mitad, la learning rate cada cierto número de epochs.
- ▶ *Exponential decay*: donde la *learning rate* sigue un decrecimiento como el de una función exponencial.



Accede a los ejercicios de autoevaluación a través del aula virtual

---

## 4.6. Regularización



Accede al vídeo «Regularización» a través del aula virtual

---

Las redes neuronales son modelos con un gran poder de representación. El gran número de parámetros y capas que podemos añadir a una red neuronal hace que sea fácil que esta aprenda demasiado bien los datos con los que estamos entrenándola. En ocasiones, es hasta posible que la red sea capaz de memorizar perfectamente un conjunto de datos de entrenamiento, llegando a clasificarlos a la perfección (100 % de *accuracy*). Sin embargo, al hacer esto, el modelo pierde capacidad de generalización y, al ser evaluado sobre un conjunto de datos no vistos durante el entrenamiento o test set, veremos una baja capacidad de predicción.

Este fenómeno se conoce como ***overfitting*** y es un fenómeno común en *machine learning*: el algoritmo está modelando demasiado bien los datos de entrenamiento mientras que pierde capacidad de generalización en datos no vistos. Más que aprender las características generales de un dataset, el algoritmo está fijándose y aprendiendo los detalles particulares de los datos con los que es entrenado.

El fenómeno puede apreciarse fácilmente en la siguiente gráfica donde, durante el entrenamiento, se van obteniendo los *prediction error* tanto para los datos de entrenamiento como para un test set que el algoritmo no está viendo. Según avanza el entrenamiento, tanto el *training error* como el *test error* disminuyen. Sin embargo, llega un punto en el que el error sobre los datos de entrenamiento sigue bajando mientras que el test error empieza a aumentar. En ese momento, se dice que el modelo está *overfitting* y perdiendo capacidad de generalización.

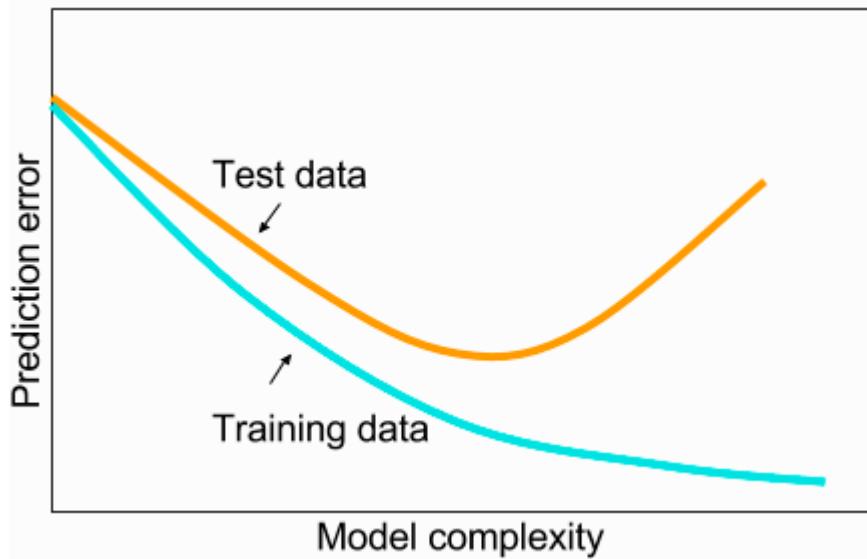


Figura 47. Overfitting.

Fuente: [http://gluon.mxnet.io/chapter02\\_supervised-learning/regularization-scratch.html](http://gluon.mxnet.io/chapter02_supervised-learning/regularization-scratch.html)

Las técnicas de **regularización** (*regularization*) intentan solucionar este problema. Su objetivo es conseguir reducir este *gap*. Ahora veremos algunas de las técnicas de regularización más comunes.

## Regularización L2

Probablemente la forma más común de aplicar regularización. Consiste en penalizar la norma al cuadrado (de ahí el nombre de L2) de los *weights* en una red neuronal en la función de coste. Dicho de otro modo, por cada parámetro  $w$  en la red, añadimos el valor cuadrado a la función de coste multiplicado por un parámetro de regularización lambda. Por ejemplo, usando *cross entropy loss*, la función de coste quedaría así:

$$C = -\frac{1}{n} \sum_{xj} [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)] + \frac{\lambda}{2n} \sum_w w^2$$

El parámetro de regularización lambda es un hiperparámetro de la red y aumentarlo implica aumentar la regularización.

Al penalizar el cuadrado de los parámetros en la función de coste (recordemos que lo queremos minimizar) estamos favoreciendo que los parámetros tengan valores absolutos pequeños. Durante el aprendizaje, la red solo asignará valores grandes a los parámetros (que son penalizados) si al hacer esto consigue reducir la *loss* en una cantidad mayor que esta penalización.

Intuitivamente, este tipo de regularización está reduciendo el espacio de posibles soluciones. Al forzar al espacio de parámetros a tener un valor pequeño, la red neuronal es menos libre para asignar valores cualesquiera y, por lo tanto, pierde cierta capacidad de expresividad, haciendo que sea más complicado fijarse en los detalles particulares en el *training data* que llevan al *overfitting*.

En la práctica, la regularización L2 es muy utilizada gracias a su buen funcionamiento. No obstante, es importante mencionar que esta regularización solo es aplicada a los pesos  $w$  y no a los *biases*  $b$ .

## Regularización L1

La regularización L1 es muy similar a la L2, pero en vez de sumar los valores al cuadrado de los pesos, suma los valores absolutos.

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|$$

La idea es similar a la regularización L2. Sin embargo, la regularización L1 tiene la interesante capacidad de hacer que los parámetros sean *sparse* (dispersos): muchos parámetros se quedan en 0 o valores muy cercanos a 0. De este modo, las neuronas acaban utilizando solo una serie de sus inputs, siendo por tanto resistentes al ruido. En comparación, la regularización L2 tiende a asignar valores pequeños a todos los pesos.

Si bien la regularización L1 es muy común en tareas de *machine learning*, en el caso de redes neuronales la regularización L2 tiende a dar mejores resultados y es más utilizada en la práctica.

### Max norm regularization

Otra idea utilizada a menudo en *deep learning* es la de constreñir los valores de los parámetros a tener una norma vectorial menor de cierto tamaño. Por ejemplo, forzar a los vectores de parámetros a tener norma 1 o menor que 1. Esta es otra forma de reducir la libertad del modelo para encontrar posibles soluciones, actuando así como fenómeno regularizador.

### Dropout

*Dropout* es una técnica bastante moderna y que ha encontrado una gran acogida, muy utilizada en la práctica. La idea consiste en aplicar una salida 0 a un porcentaje de neuronas de la red durante el entrenamiento de manera aleatoria. De este modo, en cada *batch*, un número aleatorio de neuronas se desactivará. La probabilidad  $p$  de que una neurona se desactive es otro hiperparámetro de la red y es común utilizar valores de 0.25 o 0.5. En este último caso, significa que durante el entrenamiento la mitad de las neuronas se van desactivando en cada *batch* (nótese que las neuronas inactivas cambian en cada *batch*).

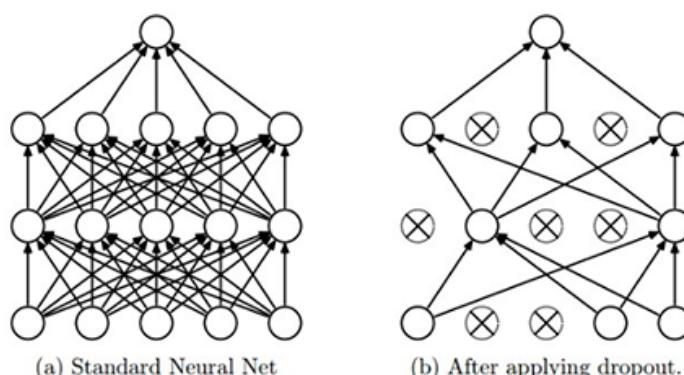


Figura 48. Efecto de aplicar *dropout*.

Fuente: <http://cs231n.github.io/>

*Dropout* impide la coadaptación de las *features*. Al forzar que ciertas neuronas tengan salida 0, la red tiene que aprender nuevas formas de correlacionar las *features*. Sin *dropout*, es posible que la red memorice la presencia continua de ciertas neuronas activadas para cierto valor de salida. Con *dropout*, la red es forzada a aprender nuevas relaciones entre neuronas, ya que muchas de las neuronas han quedado apagadas. Esto tiene un efecto regularizador, ya que se le impide a la red memorizar resultados.

Otra interpretación de *dropout* es la de que actúa como un *ensemble* de modelos. Durante el entrenamiento, estamos obteniendo en cada iteración una «nueva» red neuronal aleatoria según las neuronas que no quedan desactivadas. Según esta interpretación, estaríamos entrenando a la vez un gran número de redes neuronales ligeramente distintas.

Durante la inferencia o *test time* (cuando no estamos entrenando), *dropout* no se aplica y todas las neuronas están activadas.

### *Early stopping*

Finalmente, otra técnica muy utilizada en la práctica es la de *early stopping*. Esta técnica consiste en utilizar un *validation set* de datos que son evaluados durante el entrenamiento, pero que no son utilizados para entrenar. De este modo, podemos ver cuándo la red está empezando a *overfit*. Así, podemos parar el entrenamiento en este momento u obtener una idea de cuántas *epochs* hace falta entrenar el modelo en entrenamientos futuros.

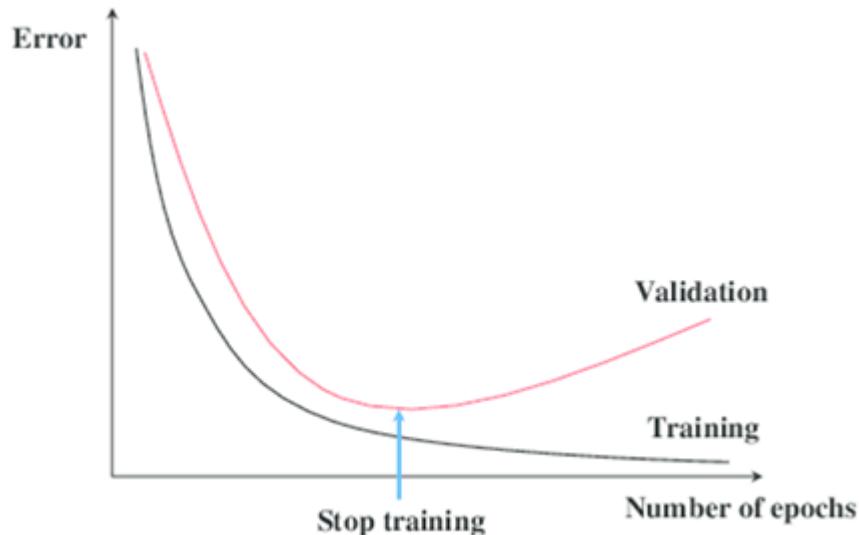


Figura 49. Efecto de aplicar *dropout*.

Fuente: [https://www.researchgate.net/figure/Early-stopping-method\\_fig3\\_283697186](https://www.researchgate.net/figure/Early-stopping-method_fig3_283697186)



Accede a los ejercicios de autoevaluación a través del aula virtual

## 4.7. Referencias bibliográficas

He, K., Zhang, X., Ren, S. y Sun, J. (2015). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. [Lugar desconocido]: Microsoft Research. Recuperado de <https://arxiv.org/pdf/1502.01852.pdf>

Krizhevsky, A., Sutskever, I. y Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems*, 25(2). DOI: 10.1145/3065386. Recuperado de <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>



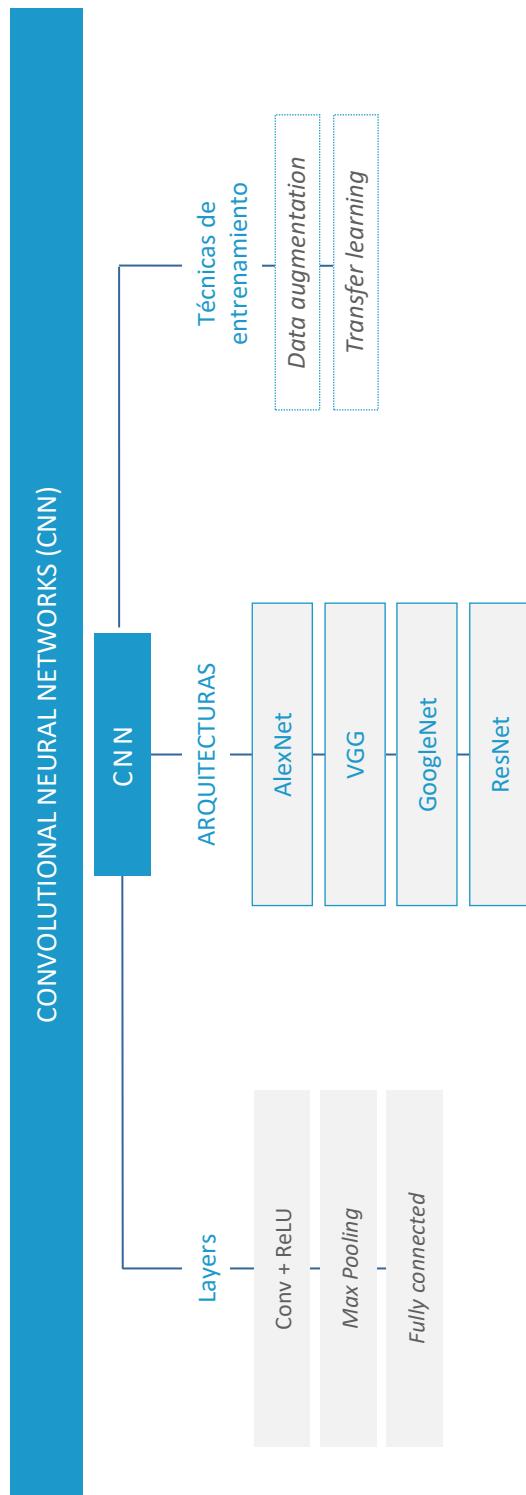
Accede al vídeo «Resumen» a través del aula virtual

# Tema 5. *Convolutional Neural Networks (CNN)*

## Índice

<b>Esquema</b>	<b>3</b>
<b>Ideas clave</b>	<b>4</b>
5.1. Introducción y objetivos	4
5.2. Introducción a las CNN	4
5.3. <i>Convolution layers</i>	11
5.4. Arquitecturas CNN para problemas de visión por computador	22
5.5. <i>Data augmentation</i>	27
5.6. <i>Transfer Learning</i>	29
5.7. Referencias bibliográficas	31

# Esquema



## 5.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

En este tema estudiaremos las *Convolutional Neural Networks* (CNN), un tipo de red neuronal muy popular para problemas de visión por computador. Veremos cómo funcionan en detalle estas redes, así como varias arquitecturas reales que han revolucionado el área de *computer vision* en los últimos años.

Es importante en este tema entender cómo funcionan las capas de una CNN y cómo se aplican a las imágenes. También es necesario comprender la dimensionalidad de entrada y de salida de estas capas y tener una idea intuitiva de cómo las redes convolucionales obtienen *features* de alto nivel. Asimismo, es importante entender cómo los conceptos de *Transfer Learning* y *Data Augmentation* ayudan a entrenar redes neuronales a partir de imágenes. Sobre las arquitecturas CNN que estudiaremos (AlexNet y VGG), no es necesario saberse de memoria sus detalles y parámetros, pero sí es importante conocer los ingredientes que las hicieron exitosas.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 5.2. Introducción a las CNN



Accede al vídeo «Introducción a las CNN» a través del aula virtual

En el tema 2 vimos cómo aplicar una red neuronal a las imágenes del dataset MNIST. En aquel caso, se colocaban todos los píxeles de la imagen concatenados en un

vector, que constituía la primera capa de nuestra red neuronal. Esta se encargaba después de clasificar las imágenes mediante el uso de una *hidden layer*.

Esto es una forma correcta de atacar un problema como MNIST. Las imágenes de MNIST son de apenas 28x28 píxeles y en blanco y negro, por lo que el tamaño de la *input layer* era de 784 elementos. Como sabemos, en *fully connected layers*, el número de parámetros crece con el producto del número de neuronas en una capa y el de la anterior (¡importante pensar mentalmente por qué esto es cierto si no lo tenemos claro!). En el caso concreto de MNIST, los números eran manejables.

Sin embargo, como sabemos, las imágenes suelen tener un número mucho mayor de píxeles y pueden estar en color, por lo que hemos de tener en cuenta los canales RGB. En este caso, una imagen de 300x300 píxeles en color tendría un total de  $300 \times 300 \times 3 = 270\,000$  valores para representarla. Si esto fuera la *input* de nuestra red neuronal, tendríamos un total de 270 000 parámetros ¡solo por cada neurona en la primera *hidden layer*! Claramente estos números son demasiados altos desde un punto de vista computacional y, además, muy probablemente llevarían a la red neuronal a sufrir de *overfitting* (recordemos que a más parámetros, más capacidad de representación de la red y, por tanto, más capacidad de «memorizar» el dataset con el que se entrena).

Las *convolutional neural networks* o **redes neuronales convolucionales**, o simplemente CNN para los amigos, intentan solucionar este problema. De manera similar a las redes neuronales convencionales, las neuronas de una red convolucional:

- ▶ Toman valores de entrada.
- ▶ Realizan un producto escalar entre sus parámetros y esos valores de entrada.
- ▶ Suman un *bias* y aplican después una *non-linearity*.
- ▶ Igualmente, tendremos una *loss function* a minimizar mediante un algoritmo de optimización.

Nada de esto cambia aquí. Sin embargo, las CNN asumen ciertas características espaciales de los *inputs* que permiten simplificar las arquitecturas, reduciendo en gran medida el número de parámetros.

Las redes neuronales convolucionales surgieron en el **contexto de la visión por computador** y son ubicuas en todo problema que implique el uso de imágenes. Como tal, este tema estará dedicado enteramente al tratamiento de imágenes, y las *inputs* a tratar serán siempre imágenes salvo que se diga lo contrario. No obstante, es importante destacar que las CNN se utilizan también en otros dominios diferentes al de visión, si bien en menor medida.

## Historia

Veamos brevemente la historia de las CNN y cómo se han erigido en la solución estándar para problemas de visión por computador. De nuevo, podemos ver cierta inspiración en el campo de la **neurociencia**.

En 1959, Hubel y Wiesel realizaron un experimento por el cual midieron los estímulos del cerebro de un gato ante ciertas formas y figuras. Esto llevó a descubrir que había ciertas áreas en el córtex relacionadas con áreas en el campo visual. Igualmente, se descubrió que las neuronas tienen cierta organización jerárquica, con neuronas sencillas encargándose de respuestas ante la orientación de la luz y otras neuronas más complejas relacionadas con el movimiento y las formas.

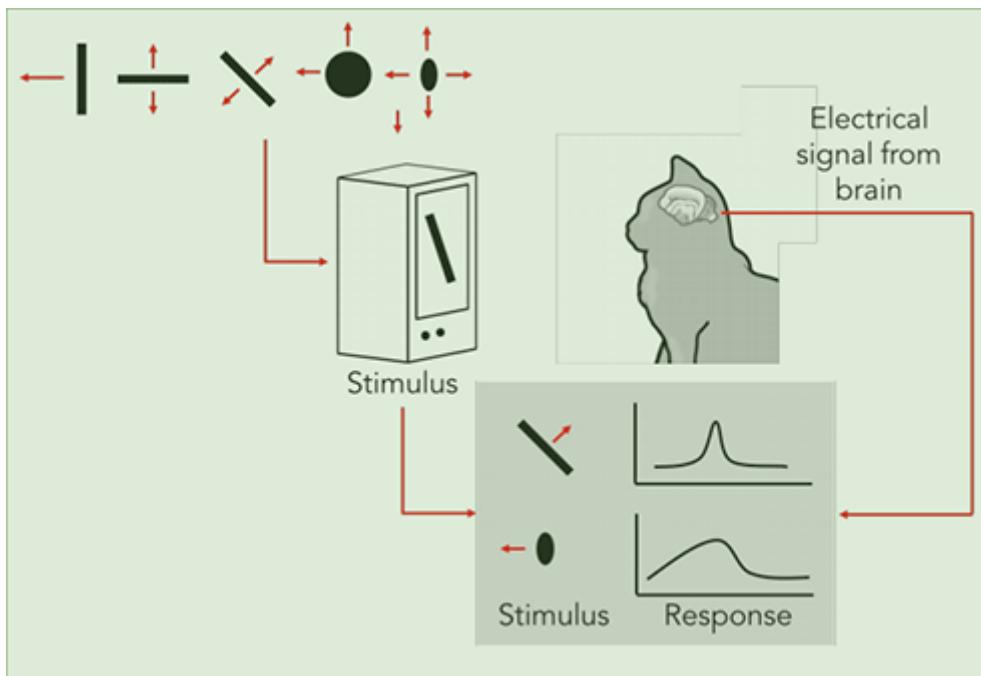


Figura 50. Reacción a estímulos en el cerebro de un gato.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture01.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture01.pdf)

En 1998, Yann LeCun, considerado uno de los padres del aprendizaje profundo moderno, introdujo el primer caso práctico de una red convolucional entrenada con *gradient descent* (LeCun, Bottou, Bengio y Haffner, 1998). Esta red era muy efectiva en reconocer dígitos para el servicio postal.

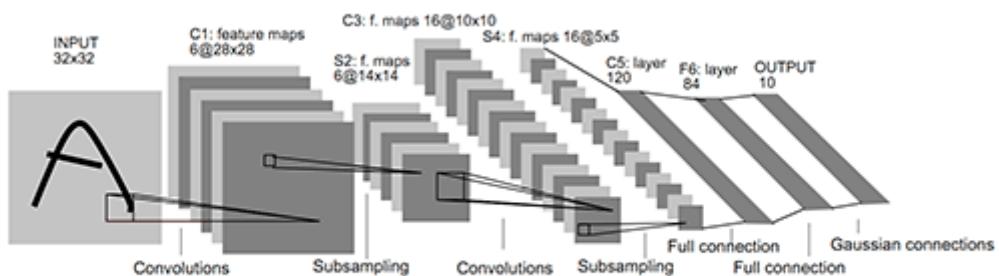


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Figura 51. Arquitectura LeNet-5.

Fuente: [https://world4jason.gitbooks.io/research-](https://world4jason.gitbooks.io/research-log/content/deepLearning/CNN/Model%20ImgNet/lenet.html)

[log/content/deepLearning/CNN/Model%20ImgNet/lenet.html](https://world4jason.gitbooks.io/research-log/content/deepLearning/CNN/Model%20ImgNet/lenet.html)

Sin embargo, la verdadera explosión en el uso de las CNN vino, como ya vimos en el primer tema, cuando una red de este tipo ganó en 2012 la competición de clasificación de imágenes ImageNet por un gran margen. Esta CNN era más profunda

que las vistas hasta el momento y presentaba una estrategia de entrenamiento mucho más efectiva, utilizando varios de los elementos que hemos visto en los temas anteriores. Asimismo, se emplearon GPU para una mayor velocidad de entrenamiento.

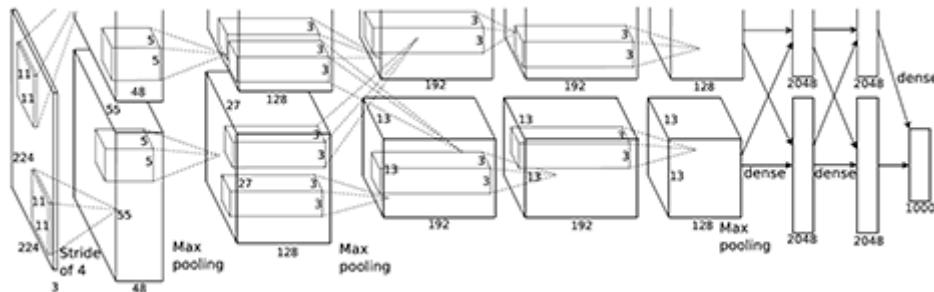


Figura 52. Arquitectura AlexNet.  
Fuente: Krizhevsky, Sutskever y Hinton, 2012.

Desde ese momento, las redes convolucionales empezaron a romper récords en tareas de visión por computador, revolucionando prácticamente el campo y permitiendo nuevos sistemas de inteligencia artificial impensables hasta el momento.

### Ejemplos de uso de redes convolucionales en la actualidad

Veamos ahora varios de los problemas que están siendo resueltos con CNN en la actualidad.

#### La clasificación de imágenes

En primer lugar, el problema que lo empezó todo: la clasificación de imágenes. Aplicando una CNN a problemas de clasificación, podemos saber qué objeto o categoría tiene una imagen:



Figura 53. Clasificación de imágenes mediante CNN.

Fuente: Krizhevsky, Sutskever y Hinton, 2012.

### Imágenes similares

Las CNN pueden utilizarse en el campo de *image retrieval* para obtener imágenes similares a otras imágenes. Las *features* aprendidas por la red convolucional pueden ser utilizadas para buscar similitudes entre aquellas.

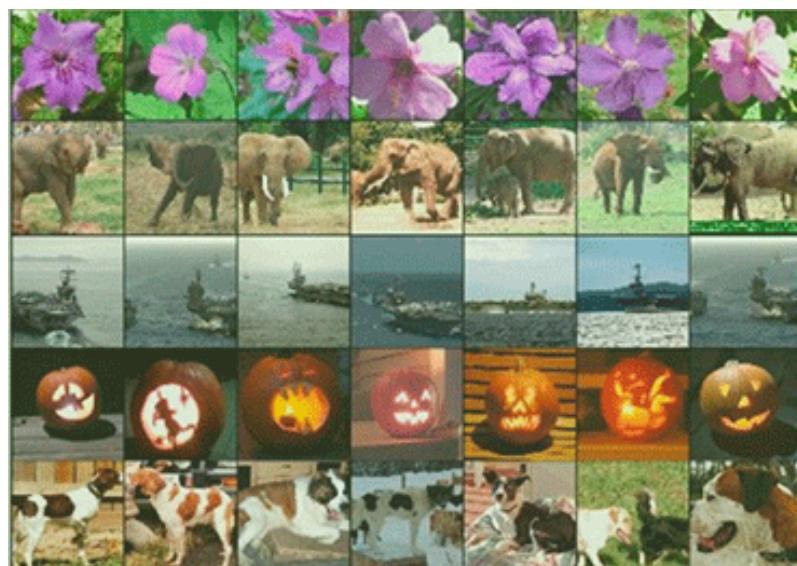


Figura 54. Búsqueda de imágenes similares mediante CNN.

Fuente: Krizhevsky, Sutskever y Hinton, 2012.

## Detección de objetos

Otra área donde las CNN han tenido gran éxito es en la detección de objetos (*object detection*). El problema consiste en ser capaz de localizar y situar objetos en imágenes.

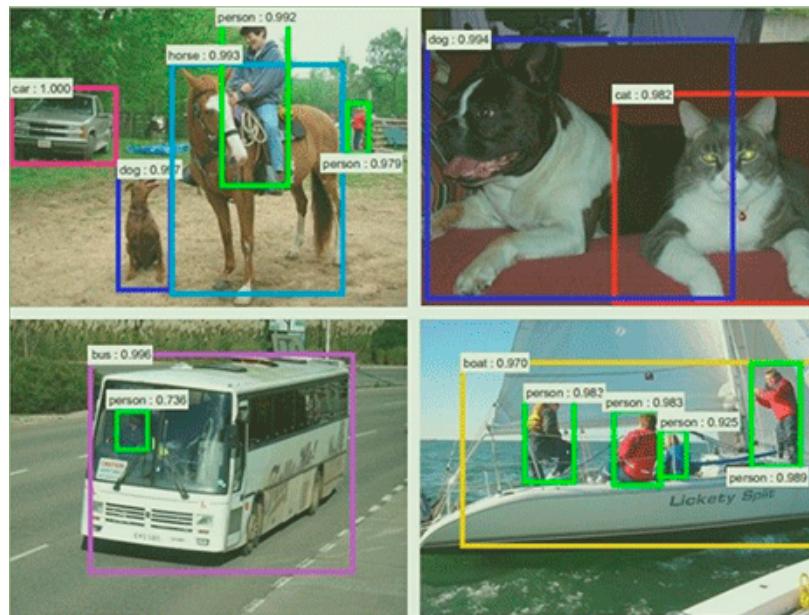


Figura 55. Detección de objetos mediante CNN.

Fuente: Ren, He, Girshick, y Sun, 2015.

## Segmentación

Del mismo modo, el problema de *segmentation* también está siendo tratado con CNN. En resumidas cuentas, el objetivo es separar la imagen en todos sus constituyentes píxel a píxel.

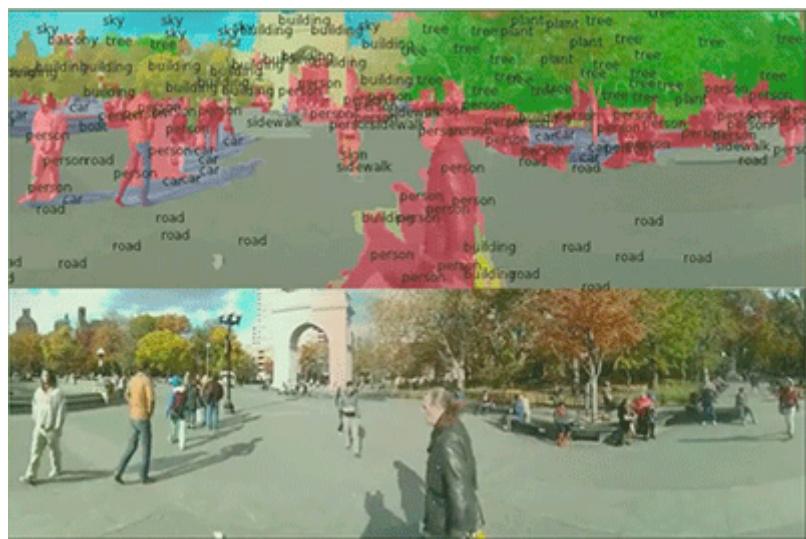


Figura 56. Segmentation mediante CNN.

Fuente: Farabet, Couprie, Najman y LeCun, 2012.

Con ejemplos como los vistos hasta ahora, queda bastante claro por qué las CNN son una de las tecnologías clave para los *self-driving cars* o coches sin conductor.

El número de aplicaciones no se reduce solo a las vistas hasta ahora. Ciertamente, casi cualquier problema de inteligencia artificial donde estén presentes imágenes o vídeos es un gran candidato a ser tratado con redes convolucionales.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 5.3. Convolution layers



Accede al vídeo «*Convolution layers*» a través del aula virtual

Las CNN utilizan distintos tipos de capas o *layers*. La capa más importante, y la que da nombre a la red, es la **capa convolucional**. Esta *layer* funciona a partir de unos filtros de tres dimensiones de pequeño tamaño, que van desplazándose por la imagen obteniendo las salidas de la capa.

En la siguiente imagen podemos observar uno de estos filtros. La imagen en este caso tiene un tamaño de 32x32x3, mientras que el filtro es más pequeño, 5x5x3. **Los filtros siempre tienen la misma profundidad (*depth*) que la imagen** (en este caso, 3), ya que se desplazarán a través de la primera y segunda dimensión.

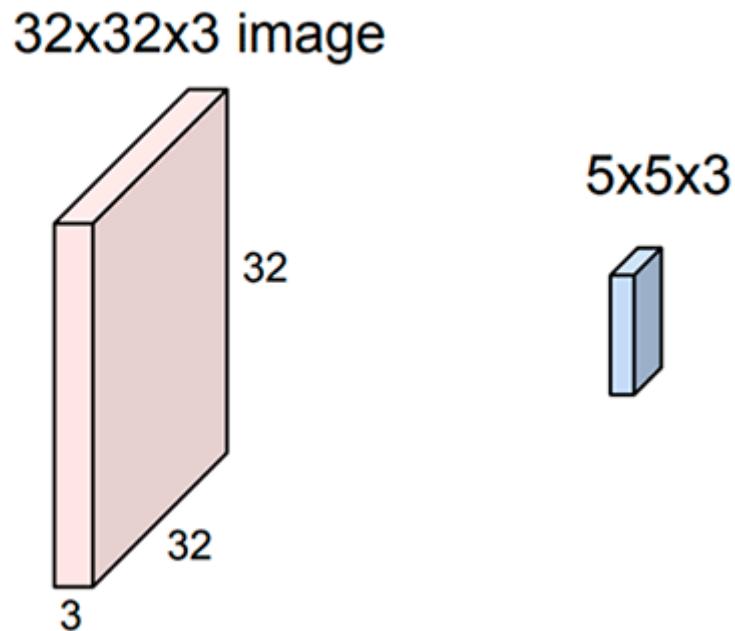


Figura 57. Imagen vista como un volumen en 3 dimensiones (la tercera dimensión corresponde a los canales RGB). A su derecha, un filtro de tamaño 5x5x3.

Fuente: <http://cs231n.github.io/>

En la figura 9, el filtro se mueve por todas las posiciones posibles en la imagen (yendo primero de izquierda a derecha y luego «pasando» a la siguiente línea), y por cada posición obtenemos una activación o un valor de salida. La idea aquí es que el filtro va recorriendo la imagen y obteniendo *features* relevantes. Como se ve, la activación resultante (*activation map*) pasa de tener un tamaño 32x32 a 28x28.

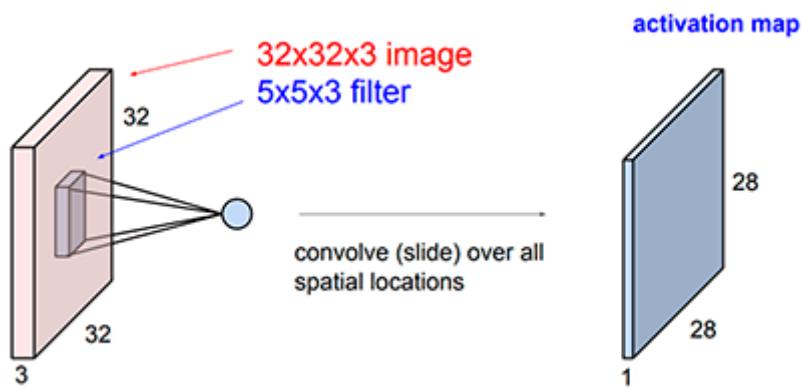


Figura 58. Convolución a partir de la imagen.

Fuente: <http://cs231n.github.io/>

El valor de salida por cada convolución aplicada con un filtro se obtiene mediante la multiplicación de todos los elementos de la imagen en los que el filtro está actuando por cada parámetro correspondiente del filtro. El filtro, de tamaño  $5 \times 5 \times 3$ , se aplica sobre un área de tamaño  $5 \times 5 \times 3$  en la imagen, por tanto, multiplicamos cada número de la imagen con el peso correspondiente del filtro.

Otra forma de verlo es pensar que concatenamos los  $5 \times 5 \times 3 = 75$  elementos del filtro en un vector de parámetros  $w$  y hacemos el producto escalar con los  $5 \times 5 \times 3 = 75$  elementos de la imagen sobre los que el filtro está siendo aplicado. Con esto podemos ver por qué el filtro tiene que tener la misma profundidad que la imagen, ya que si no, no podríamos aplicar esta operación. Igualmente, a esta multiplicación de valores se le aplica del mismo modo un término *bias*  $b$ .

Es importante comprender que los parámetros de la red convolucional, los pesos  $w$  que aprenderemos durante el entrenamiento, están en los filtros en el caso de las capas convolucionales.

No es común aplicar solo un filtro en una capa convolucional; lo más normal es tener varios filtros para obtener más *features* en cada posición de la imagen. La idea es que cada uno de estos filtros obtenga ciertas características de la imagen que serán importantes a la hora de obtener una representación suficientemente expresiva de

la misma. En el siguiente diagrama, vemos cómo se aplica un segundo filtro y se obtiene un nuevo «mapa de activación» de tamaño 28x28x1.

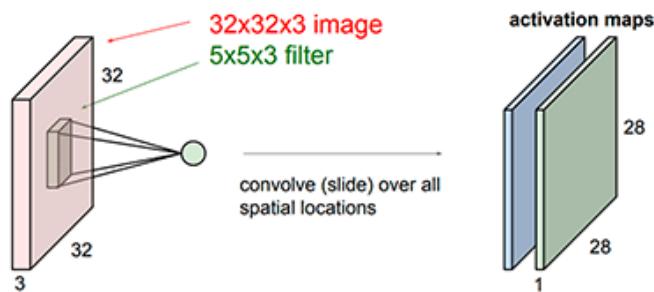


Figura 59. Aplicando un segundo filtro a la convolución.

Fuente: <http://cs231n.github.io/>

Del mismo modo, aplicando más filtros acabamos obteniendo una nueva representación en tres dimensiones de la imagen, conservando sus características espaciales y, a la vez, añadiendo más profundidad con las nuevas *features* calculadas a partir de los filtros.

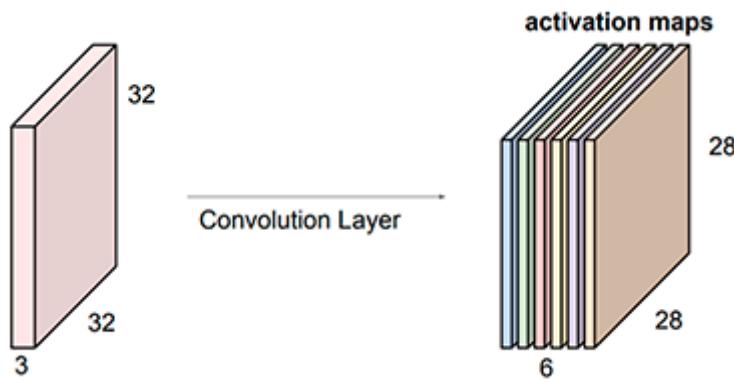


Figura 60. Las redes convolucionales aplican un gran número de filtros para obtener nuevas *features*.

Fuente: <http://cs231n.github.io/>

Es fácil ver cómo hay muchos menos parámetros en estas redes (la suma de los parámetros de cada filtro) para tratar una imagen en comparación con una *fully connected layer*. Dejamos como ejercicio para el alumno hacer esta comprobación.

## Creando una red convolucional a partir de las capas convolucionales

Como vemos, las capas convolucionales aplican distintos filtros sobre un volumen de entrada y crean nuevos volúmenes, por lo que las propiedades espaciales de la imagen se mantienen. Lo más común es aplicar capas convolucionales seguidas de funciones de activación ReLU. La unidad ReLU se aplicaría sobre cada valor de activación salido de un filtro aplicado sobre un área. Dicho de otro modo, por cada valor del volumen de salida aplicamos la *non-linearity*.

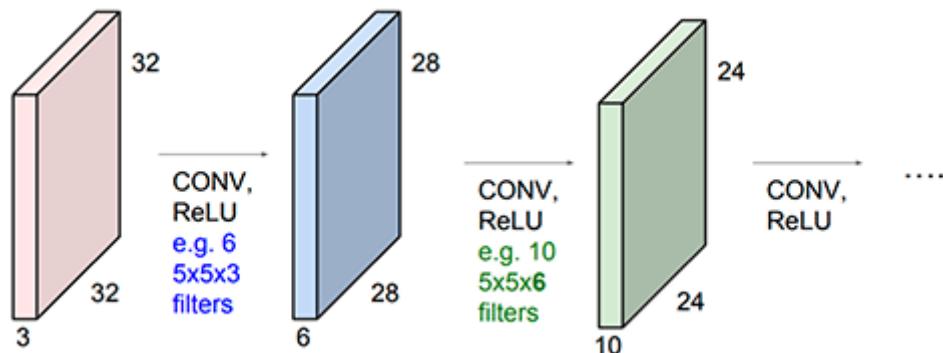


Figura 61. Las CNN suelen estar compuestas por capas convolucionales + ReLUs aplicadas de manera sucesiva.

Fuente: <http://cs231n.github.io/>

La idea aquí es que, al configurar una red de esta forma, las distintas capas van obteniendo una representación jerárquica de las *features*, con las primeras capas reconociendo elementos más simples en una imagen y las siguientes obteniendo representaciones de más alto nivel a partir de estos elementos simples. En la siguiente imagen, podemos ver un ejemplo típico de una red convolucional (con sus distintas capas) aplicada a un problema de clasificación. La capa de *pooling*, POOL en la imagen, será vista más adelante en este tema.

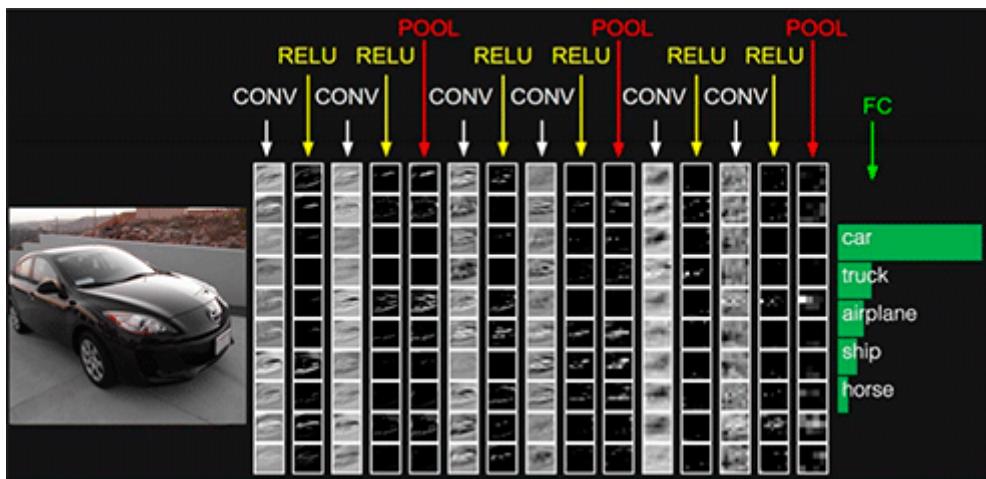


Figura 62. CNN aplicada a un problema de clasificación vista capa a capa.

Fuente: <http://cs231n.github.io/>

### Stride y zero-padding

#### Stride

Veamos unos ejemplos más claros de cómo se realizan las convoluciones espacialmente. Aparte del tamaño del filtro, un elemento importante es el *stride*, que **indica cuánto se desplaza el filtro a través de la imagen**.

Por ejemplo, con un *stride* de tamaño 1, podemos ver en la siguiente imagen cómo un filtro  $3 \times 3$  se mueve desplazándose un cuadro a la derecha (de arriba a abajo sería similar). En este ejemplo no estamos considerando la tercera dimensión (*depth*) del *input* ni del filtro que, como sabemos, tienen que coincidir.

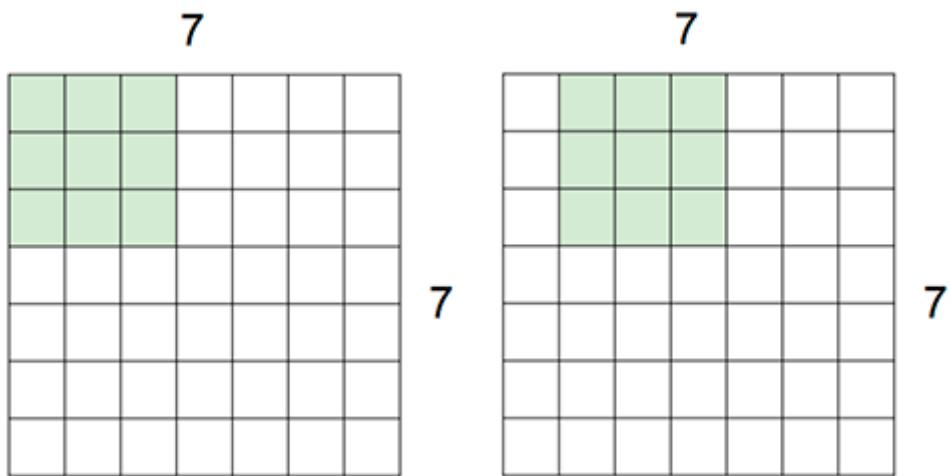


Figura 63. Movimiento de un filtro de izquierda a derecha con stride 1.

Fuente: <http://cs231n.github.io/>

De manera equivalente, un *stride* de 2 haría que el filtro se desplazara dos cuadritos a la derecha. Aumentar el tamaño del *stride* hace que las *features* que obtenemos en la red convolucional se fijen en áreas más distantes de la imagen, pero también implican una **reducción en la dimensionalidad**.

Si:

- ▶  $N$  es el tamaño del *input* para una imagen NxN.
- ▶  $F$  el tamaño del filtro.
- ▶  $S$  el tamaño del *stride*.

Tenemos que la dimensión resultante de cada lado de nuestro *output* sería:

$$[(N - F) / S] + 1.$$

### ***Zero-padding***

Para evitar la reducción de dimensionalidad, es común aplicarlo y que el volumen resultante sea del mismo tamaño que el *input*. El *zero-padding* consiste en **añadir ceros a los lados para que las dimensiones cuadren**.

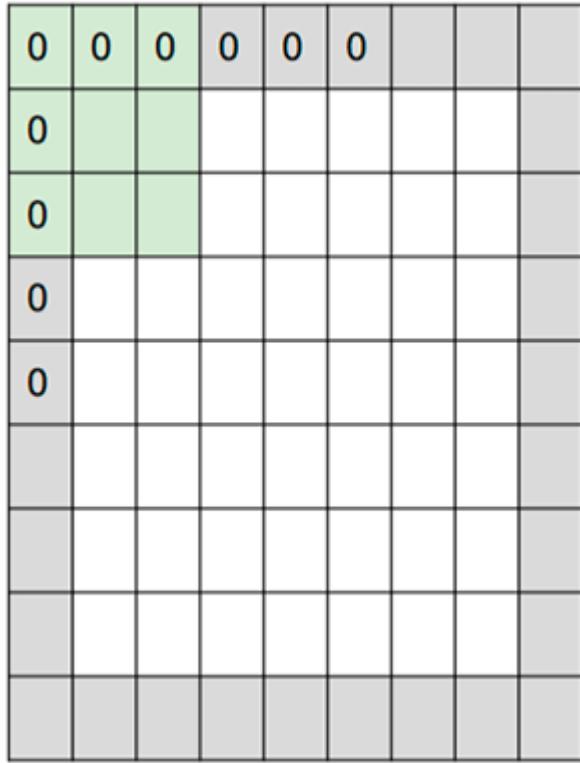


Figura 64. *Zero-padding* en una imagen.

Fuente: <http://cs231n.github.io/>

Con *zero-padding*, la fórmula anterior cambia un poco. En general, una capa convolucional con *input* y un volumen de tamaño  $W_1 \times H_1 \times D_1$ :

- ▶ Tiene cuatro hiperparámetros:
  - Número de filtros  $K$ .
  - Tamaño de filtro  $F$ .
  - Stride  $S$ .
  - Cantidad de *zero-padding*  $P$ .
- ▶ Produce un nuevo volumen  $W_2 \times H_2 \times D_2$  donde:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$
  - $D_2 = K$

## Capas *Max pooling*

Las *max pooling layers* son un tipo de capa que está presente en una gran cantidad de arquitecturas CNN. La idea es **reducir (*downsample*) las representaciones** obtenidas de manera que estas se hagan más pequeñas y sean más manejables computacionalmente, reduciendo el número de parámetros necesarios y, por tanto, ayudando también a reducir el *overfitting*.

Las capas *max pooling* actúan de manera independiente sobre cada nivel de profundidad del volumen de entrada y reducen su tamaño mediante la aplicación de máximos.

La forma más común de utilizar *max pooling* es mediante filtros de tamaño 2x2 y *stride* 2. De este modo, la salida de la capa *pooling* por cada filtro de tamaño 2x2 es el máximo valor en el recuadro donde se aplica. En la figura 16 podemos ver con mayor claridad cómo al aplicar *pooling* el tamaño se reduce en un 75 %.

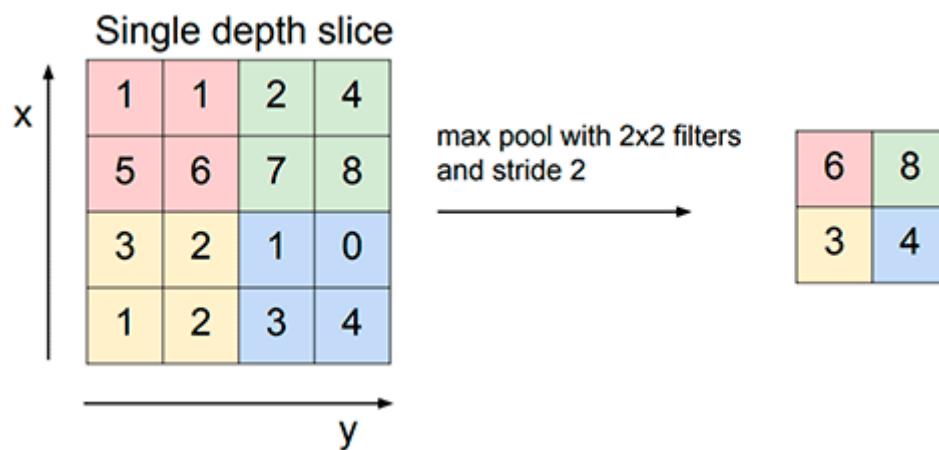


Figura 65. *Max pooling* aplicado en un solo nivel de profundidad.

Fuente: <http://cs231n.github.io/>

Las capas *max pooling* no afectan la profundidad del volumen de entrada. La reducción de la representación ocurre en el **plano espacial**, pero no en el número de *features* distintas obtenidas en cada posición.

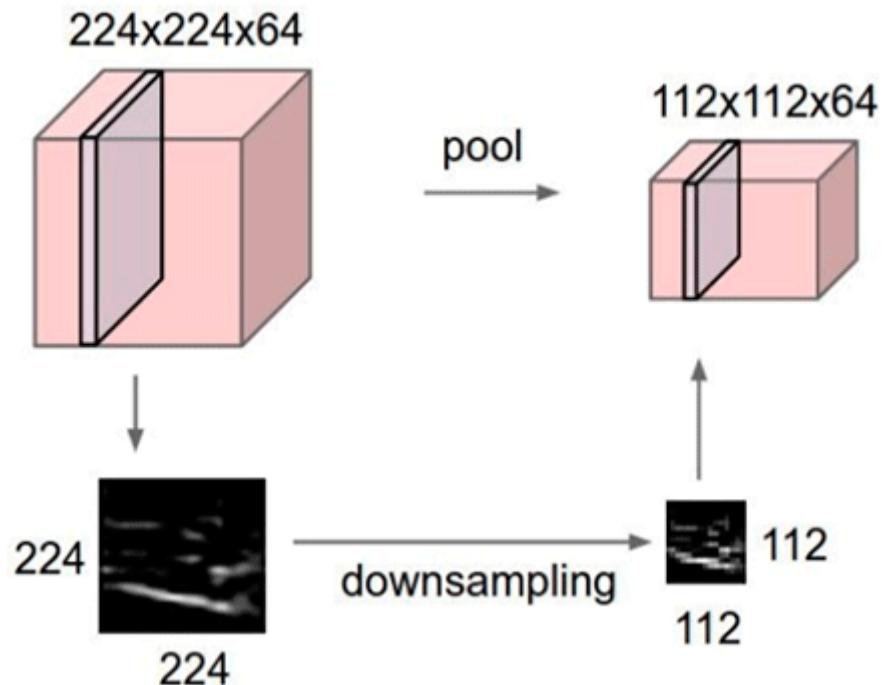


Figura 66. *Max pooling* aplicado en un solo nivel de profundidad.

Fuente: <http://cs231n.github.io/>

Las capas *max pooling* se ubican normalmente entre capas convolucionales, como vimos en una imagen anterior. Una interpretación del funcionamiento mediante la toma de máximos es que estas capas seleccionan las activaciones más importantes en cada región de una imagen o *input*. Otras posibles estrategias de reducción de dimensionalidad, por ejemplo, obtener la media de valores en el filtro en vez del máximo, no resultan tan efectivas en la práctica.

En resumen, una capa de *max pooling* aplicada a un *input* de tamaño  $W_1 \times H_1 \times D_1$ :

- ▶ Tiene dos hiperparámetros:
  - Tamaño de filtro o *pool F*.
  - Stride *S*.
- ▶ Produce un nuevo volumen  $W_2 \times H_2 \times D_2$  donde:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$

- $D_2 = D_1$

- ▶ No añade parámetros a la red neuronal ya que calcula una función fija (max).

### Fully connected layers

La última capa de una red convolucional para problemas de clasificación es una *fully connected layer*, ya que necesitamos una neurona de salida para cada clase. Normalmente, esto se hace mediante una capa *softmax*. Muchas CNN llevan varias *fully connected layers* como últimas capas para obtener las representaciones finales después de las capas de *convolutions + pooling*.

Volviendo a la imagen que vimos anteriormente (figura 13), una arquitectura CNN común sería una serie de capas convolucionales seguidas de ReLU y *max pooling layers*. La última o últimas capas serían *fully connected layers*.

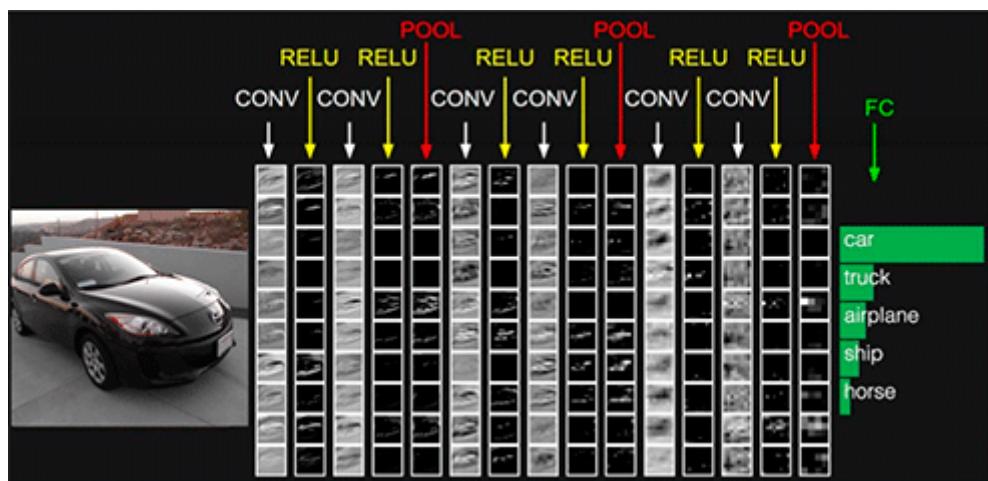


Figura 67. CNN aplicada a un problema de clasificación vista capa a capa.

Fuente: <http://cs231n.github.io/>



Accede a los ejercicios de autoevaluación a través del aula virtual

## 5.4. Arquitecturas CNN para problemas de visión por computador



Accede al vídeo «Arquitecturas CNN para problemas de visión por computador» a través del aula virtual

Una vez vistos los ingredientes básicos del funcionamiento de las CNN, veremos aquí algunas de las arquitecturas más conocidas con más detalle. Estas arquitecturas han ido poco a poco obteniendo nuevos récords en la competición ILSVRC (*ImageNet Large Scale Visual Recognition Challenge*). Esta competición evalúa algoritmos para detección de objetos y clasificación de imagen, y utiliza el famoso dataset ImageNet que consiste en 1000 categorías a clasificar.

Como ya dijimos al principio de este tema, AlexNet fue el primer modelo de aprendizaje profundo que ganó esta competición, obteniendo una gran diferencia con respecto a los ganadores de ediciones anteriores. A partir de este hito, nuevas CNN, cada vez más sofisticadas, han ido batiendo año a año el récord anterior, hasta el punto de que estas redes son capaces de resolver el problema mejor que un humano según varios *benchmarks*.

En la siguiente imagen, podemos ver los ganadores de ILSVRC desde el inicio de la competición. Puede apreciarse el salto cualitativo de AlexNet respecto a los anteriores ganadores, así como la aparición de nuevas arquitecturas CNN cada vez más profundas y efectivas:

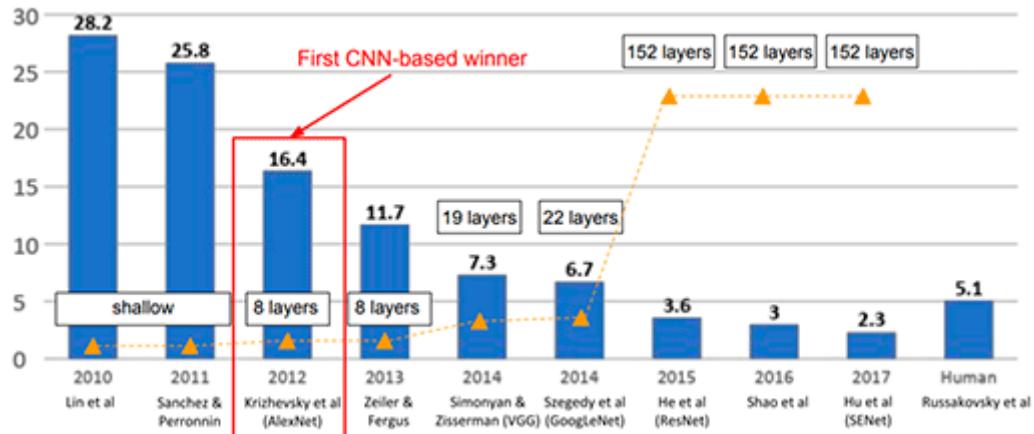


Figura 68. Ganadores de ILSVRC (2010-2017) y comparación con el rendimiento de humanos en la tarea.

Fuente: <http://cs231n.github.io/>

## AlexNet

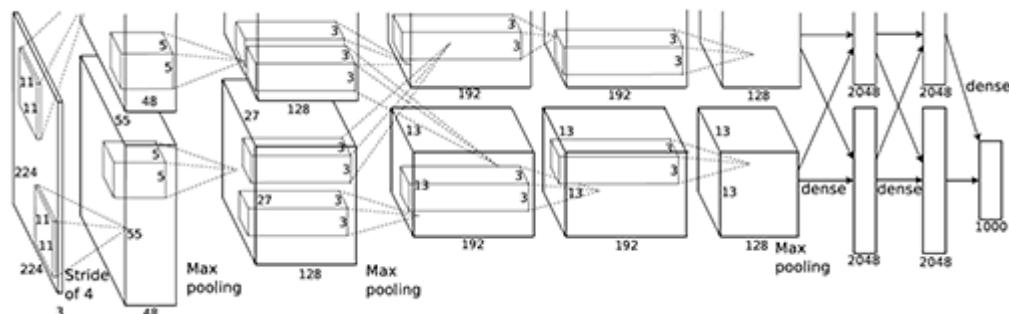


Figura 69. Arquitectura AlexNet.

Fuente: Krizhevsky, Sutskever y Hinton, 2012.

La arquitectura por capas de AlexNet es la siguiente:

- ▶ Input: imágenes 227x227x3.
- ▶ Primer bloque:
  - CONV 1 (96 11x11 filters, stride 4, pad 0).
  - MAX POOL 1 (3x3 filters, stride 2).
  - NORM 1 (Normalization layer).
- ▶ Segundo bloque:
  - CONV 2 (256 5x5 filters, stride 1, pad 2).
  - MAX POOL 2 (3x3 filters, stride 2).

- NORM 2 (Normalization layer).
- ▶ Tercer bloque:
  - CONV 3 (384 3x3 filters, stride 1, pad 1).
- ▶ Cuarto bloque:
  - CONV 4 (384 3x3 filters, stride 1, pad 1).
- ▶ Quinto bloque:
  - CONV 5 (256 3x3 filters, stride 1, pad 1).
  - MAX POOL 3 (3x3 filters, stride 2).
- ▶ Fully connected layer (FC6), 4096 neuronas.
- ▶ Fully connected layer (FC7), 4096 neuronas.
- ▶ Fully connected layer (FC8), 1000 neuronas, una por clase final a predecir.

El único elemento que no hemos visto aquí son las *normalization layers*, una capa específica de AlexNet que no es muy común en la actualidad y que consiste en realizar una normalización que según los autores dio buenos resultados en esta arquitectura.

Como vemos, una característica de AlexNet es que la profundidad de los volúmenes (dicho de otro modo, el número de filtros) va aumentando capa por capa, a la vez que el tamaño espacial se reduce mediante *max pooling*. Si bien esto no es una regla escrita, es algo muy común en arquitecturas CNN. Se puede entender intuitivamente que queremos que la red vaya obteniendo representaciones más complejas, donde los elementos de la imagen se van componiendo de una manera jerárquica.

Otro punto importante de AlexNet es que fue la primera arquitectura en utilizar la **unidad de activación ReLU**. Esta *non-linearity* es la más utilizada en la actualidad en el mundo del aprendizaje profundo y, como vemos, su origen proviene de las arquitecturas profundas para visión por computador.

Como ejercicio, sería interesante que el alumno compruebe, bloque por bloque de la arquitectura, que las dimensiones tienen sentido a partir de lo que hemos aprendido en los puntos anteriores.

Sin embargo, hay algo que resulta extraño en el esquema de la arquitectura: **¿por qué los bloques están partidos en dos trozos a lo largo de la profundidad del volumen?** Por ejemplo, los 96 filtros del primer bloque se reparten en dos partes de 48 elementos de profundidad cada una. Pues bien, esto se debe a que esta arquitectura fue entrenada utilizando GPU para acelerar el entrenamiento. Las tarjetas gráficas utilizadas solo disponían de 3GB de memoria (en la actualidad los números son mayores), por lo que era imposible mantener toda la arquitectura en una sola de ellas. De este modo, el entrenamiento se realizó distribuyendo la red en dos GPU, lo cual es una estrategia que veremos en temas posteriores.

Otros detalles interesantes sobre AlexNet son:

- ▶ Input: imágenes 227x227x3
- ▶ Se usa *dropout* con valor 0.5.
- ▶ El *batch size* es 128.
- ▶ El algoritmo de optimización es SGD con Momentum 0,9.
- ▶ El *learning rate* utilizado es 1e-2, dividido por 10 manualmente cuando el *validation error* deja de mejorar.
- ▶ Se aplica regularización L2 con peso 5e-4.
- ▶ El resultado final es un *ensemble* de 7 modelos AlexNet, lo cual permite reducir el *error rate* notablemente.

## VGGNet

VGGNet es una arquitectura similar a AlexNet que se caracteriza por ser más profunda y utilizar filtros más pequeños, de tamaño constante 3x3. Fue ganadora de ILSVRC en 2014 junto con la arquitectura GoogLeNet. Hay dos versiones de esta arquitectura, VGG16 y VGG19, con 16 y 19 capas respectivamente.

Podemos ver la arquitectura en las siguientes imágenes. Como vemos, de manera similar a AlexNet, los bloques son cada vez más profundos y las dimensiones

espaciales se reducen. Todas las convoluciones utilizan filtros 3x3 con *stride* 1 y *pad* 1, mientras que las capas *max pooling* utilizan 2x2 *pools* con *stride* 2.

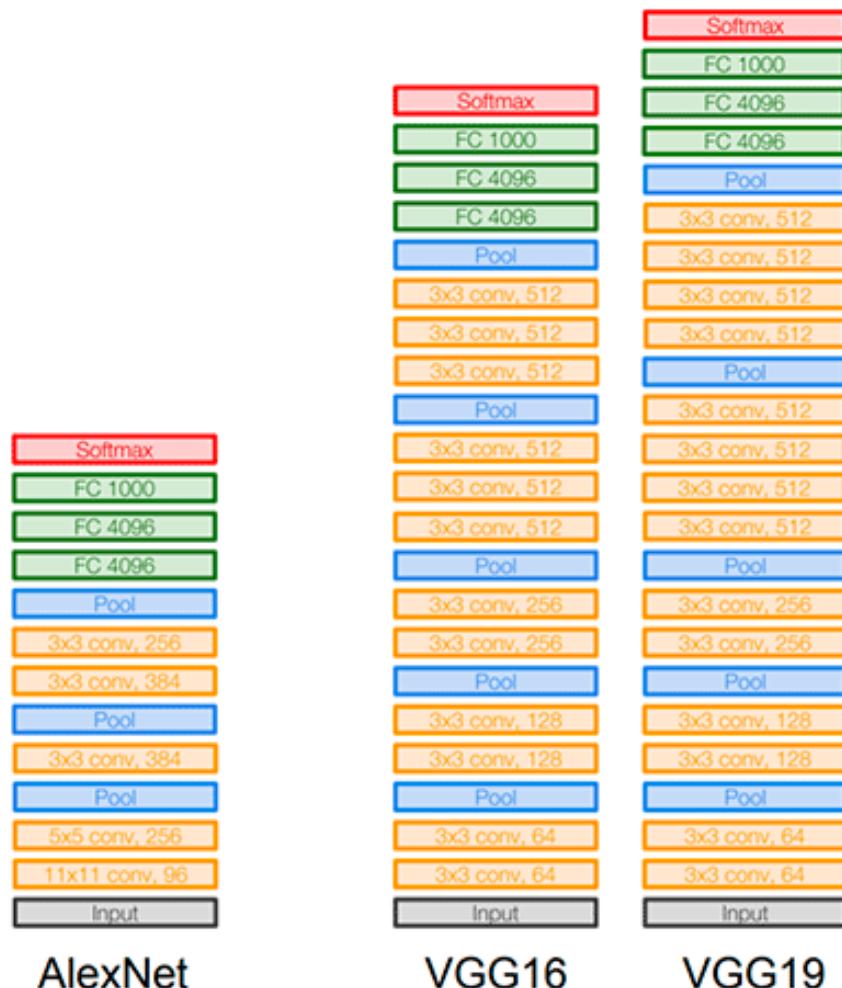


Figura 70. Arquitecturas VGG16 y VGG19 junto con AlexNet.

Fuente: <http://cs231n.github.io/>

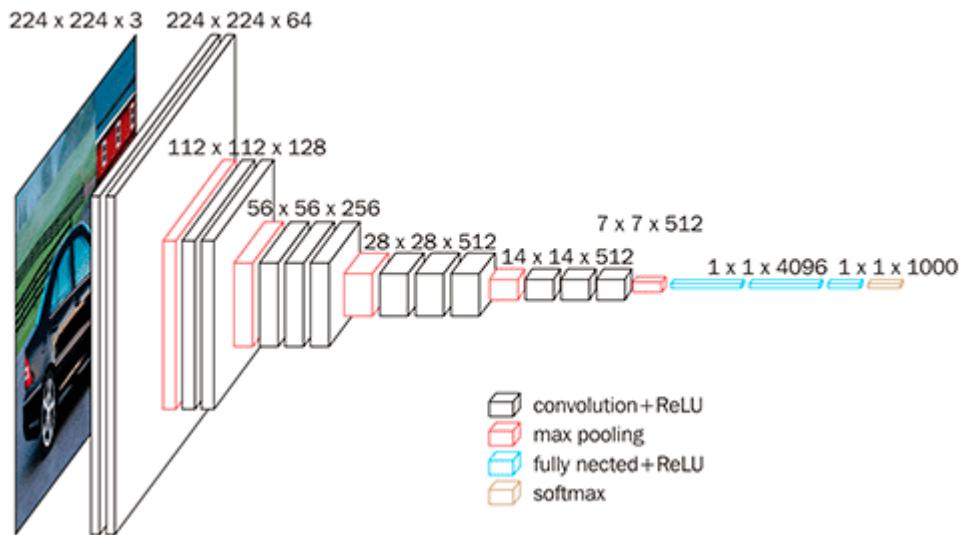


Figura 71. Arquitectura VGG.

Fuente: <https://www.safaribooksonline.com/library/view/machine-learning-with/9781786462961/21266fa5-9e3b-4f9e-b3c6-2ca27a8f8c12.xhtml>

El número de parámetros total de esta red es de aproximadamente 138 millones, en comparación con los aproximadamente 60 millones de AlexNet.

Un ejercicio interesante para el alumno sería obtener estos números.

Este enorme tamaño de VGG la hace un poco más compleja de entrenar y utilizar en la práctica. Arquitecturas más recientes tienden a eliminar las *fully connected layers* del final, lo cual no parece afectar mucho el rendimiento.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 5.5. Data augmentation



Accede al vídeo «*Data augmentation*» a través del aula virtual

Una técnica muy aplicada en problemas de visión por computador es la de *data augmentation*. Consiste en realizar transformaciones o pequeñas perturbaciones

aleatorias de la imagen de manera que obtenemos nuevas imágenes con las que entrenar, pero utilizando la misma clase o *label*. Esto permite que nuestra red vea nuevos (pero ligeramente distintos) ejemplos que consisten en el mismo objeto que queremos clasificar, obteniendo así nuevos puntos de vista y permitiendo reducir la cantidad de datos que necesitamos para entrenar.

Un ejemplo de una modificación de la imagen sería utilizar la **imagen simétrica**.

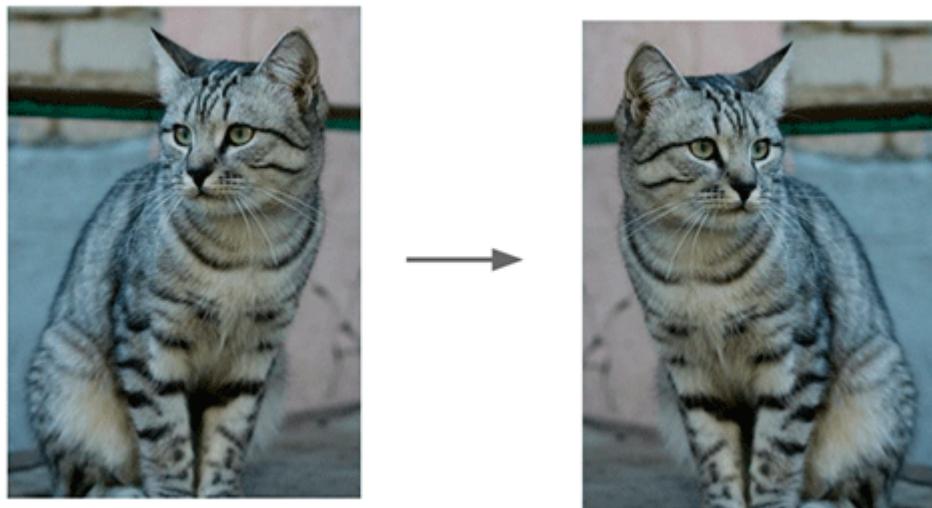


Figura 72. Ejemplo de imagen simétrica.

Fuente: <http://cs231n.github.io/>

Otra opción es cambiar aleatoriamente **el contraste y el brillo** de la imagen:

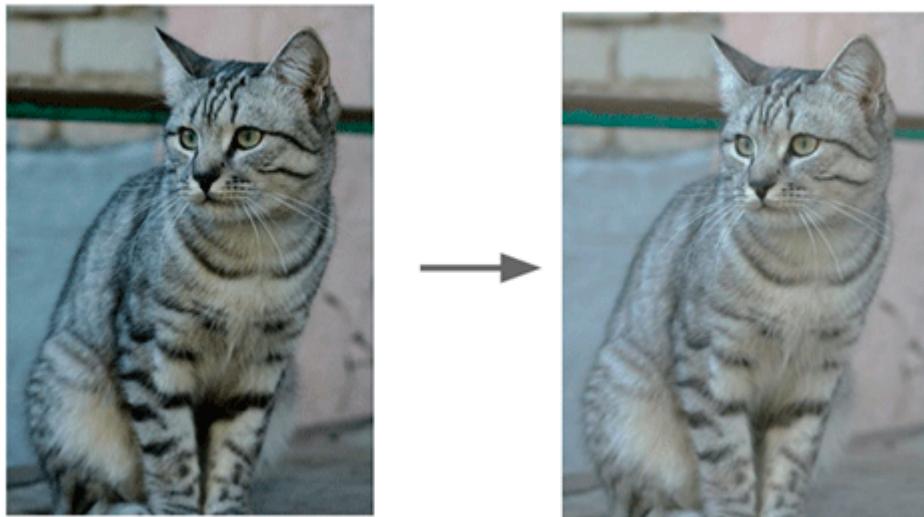


Figura 73. Ejemplo de modificación de contraste en una imagen.

Fuente: <http://cs231n.github.io/>

En general, cualquier modificación donde la imagen siga teniendo la misma clase es válida. Por ejemplo: rotaciones, translaciones, estiramientos de la imagen... Otra forma muy común de aplicar *data augmentation* es mediante la obtención aleatoria de **pequeñas partes recortadas de la imagen**.

El proceso de obtener más *training data* mediante *data augmentation* puede también verse como una forma de **regularización**. Al añadir cierta aleatoriedad y variaciones en las imágenes, estamos impidiendo en cierta medida que la red aprenda ciñéndose a elementos particulares de las imágenes originales.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 5.6. Transfer Learning



Accede al vídeo «*Transfer Learning*» a través del aula virtual

Una situación común a la hora de entrenar modelos de visión por computador es que no contamos con una cantidad de datos suficiente para entrenar una CNN. Como hemos visto, las CNN son modelos complejos con muchos parámetros y, por lo tanto, necesitan una gran cantidad de datos para ser entrenados con éxito. Una arquitectura tan grande aplicada a pocos datos llevará con gran probabilidad a *overfitting*.

Es muy difícil y costoso hacerse con una cantidad de datos como la que tiene ImageNet. *Transfer learning* es una técnica que intenta mitigar este problema mediante la **transferencia de lo aprendido con grandes datasets a problemas relativamente similares**. La idea consiste en utilizar un modelo ya entrenado (por ejemplo, VGG entrenado en ImageNet) y utilizar el *training data* de nuestro problema modificando solo los parámetros de las últimas capas del modelo, congelando el resto. De este modo, el aprendizaje que se hizo sobre el problema original se transfiere en parte al nuevo problema, intentando mantener elementos que probablemente son comunes a ambos (recordemos que en las primeras capas las redes neuronales tienden a aprender representaciones sencillas como formas e interacciones básicas).

Por ejemplo, si queremos entrenar un clasificador de razas de gatos con 10 clases, podemos coger una red entrenada con las 1000 clases de ImageNet y volver a entrenar reinicializando la última capa y permitiendo que solo esta pueda ser entrenada. De este modo, únicamente las combinaciones de representaciones de alto nivel cambian, adaptándose al nuevo problema.

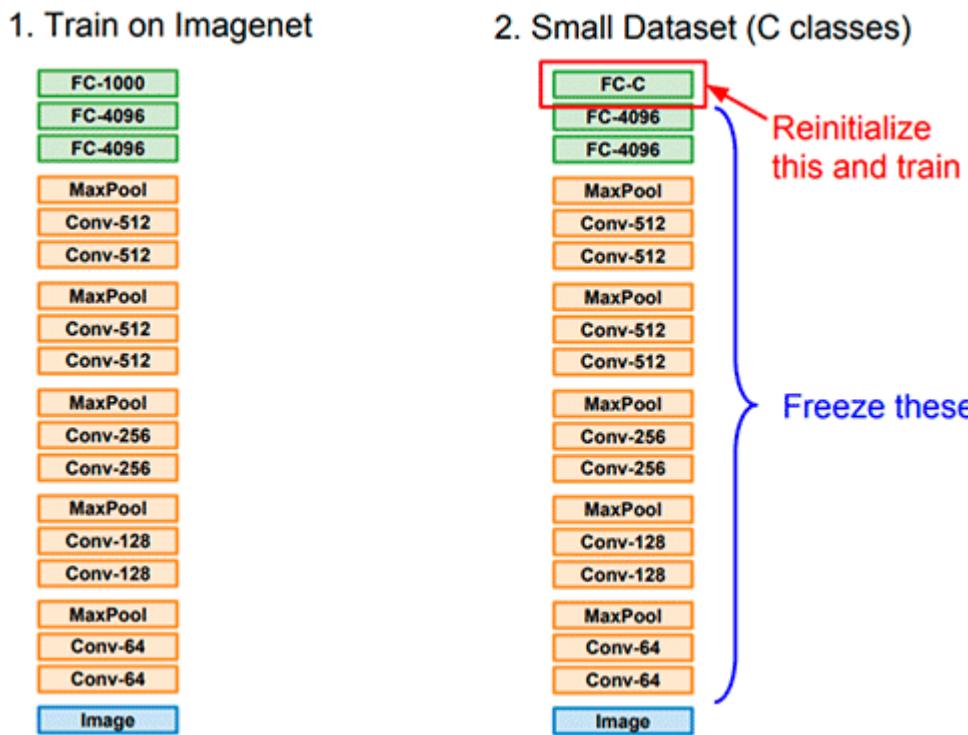


Figura 74. *Transfer learning*.

Fuente: <http://cs231n.github.io/>

*Transfer learning* es extremadamente común en el mundo del *deep learning*. No hay realmente una fórmula exacta a la hora de aplicarlo. Según la cantidad de datos de la que dispongamos, podemos reinicializar y entrenar un mayor número de capas del modelo original. Por otro lado, la efectividad del *transfer learning* puede verse comprometida si el problema a tratar es muy distinto del problema con el que se entrenó la red original.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 5.7. Referencias bibliográficas

Farabet, C., Couprie, C., Najman, L. y LeCun, Y. (2012). Learning Hierarchical Features for Scene Labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8), 1915-1929. DOI: 10.1109/TPAMI.2012.231. Recuperado de <http://yann.lecun.com/exdb/publis/pdf/farabet-pami-13.pdf>

Krizhevsky, A., Sutskever, I. y Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems*, 25(2). DOI: 10.1145/3065386. Recuperado de <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>

LeCun, Y., Bottou, L., Bengio, Y. y Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324. DOI: 10.1109/5.726791. Recuperado de  
[http://vision.stanford.edu/cs598\\_spring07/papers/Lecun98.pdf](http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf)

Ren, S., He, K., Girshick, R. y Sun. J. (2017). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6), 1137-1149. DOI: 10.1109/TPAMI.2016.2577031. Recuperado de <https://arxiv.org/pdf/1506.01497.pdf>



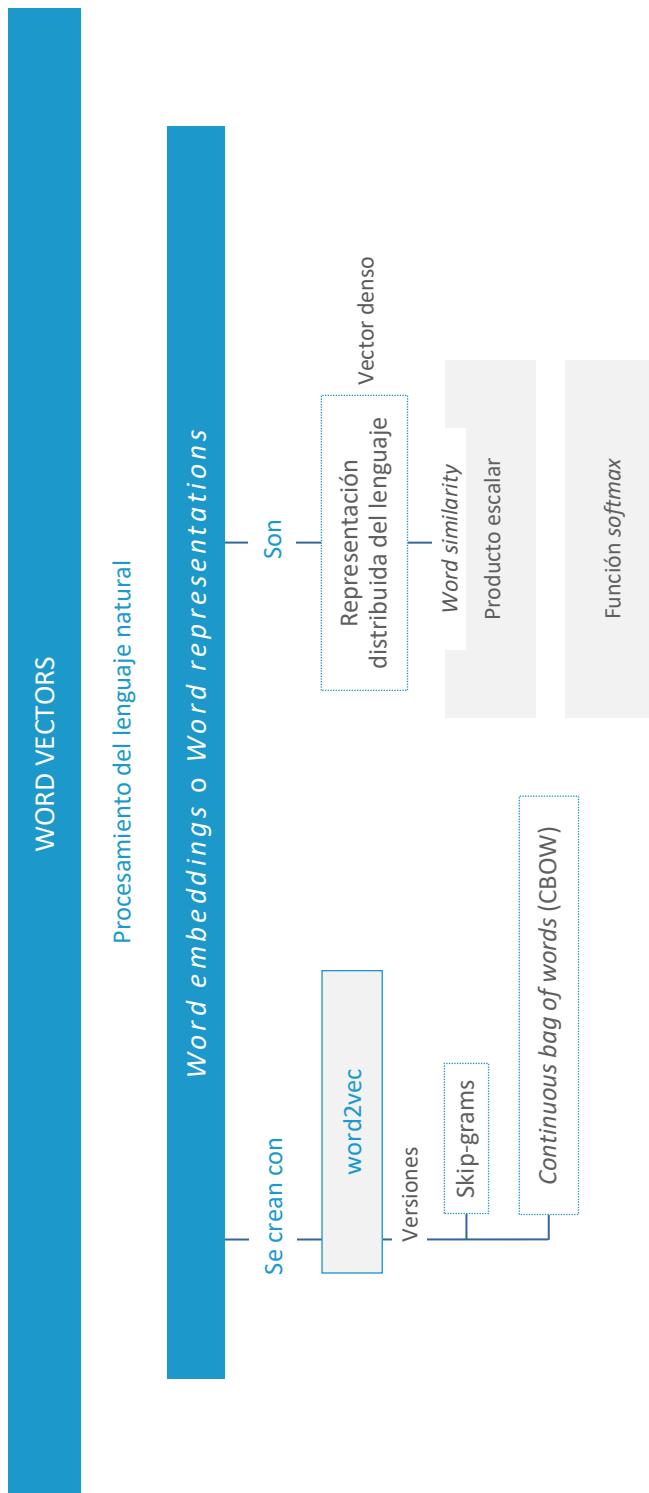
Accede al vídeo «Resumen» a través del aula virtual

# Tema 6. *Word Vectors*

## Índice

Esquema	3
Ideas clave	4
6.1. Introducción y objetivos	4
6.2. Representaciones del lenguaje	4
6.3. Word2Vec	7
6.4. Referencias bibliográficas	17

# Esquema



# Ideas clave

## 6.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

En este tema nos adentraremos en el mundo del PLN o procesamiento del lenguaje natural (NLP en inglés: *natural language processing*). En particular, veremos un tipo de representación de palabras llamado *word vectors*, que se ha hecho muy popular en los últimos años y ha impulsado la utilización del *deep learning* en problemas de lenguaje natural, así como un algoritmo para obtener estos vectores, **word2vec**.

Es importante comprender en este tema las distintas formas de representación del lenguaje y por qué los *word vectors* tienen más capacidad de representación que otros métodos anteriores. También es importante entender el funcionamiento de word2vec y las propiedades de los *word vectors* resultantes.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 6.2. Representaciones del lenguaje



Accede al vídeo «Representaciones del lenguaje» a través del aula virtual

El lenguaje natural ha sido siempre un problema esquivo para la inteligencia artificial. Los matices y ambigüedades presentes en la comunicación humana a través del lenguaje han hecho que los sistemas de inteligencia artificial no sean capaces en muchas instancias de entender cosas que son triviales para un humano.

Uno de los problemas fundamentales es la representación en un ordenador del significado de una palabra. Una solución clásica al problema ha sido la creación de bases de datos con palabras agrupadas por significado y reglas que definen las relaciones semánticas entre ellas. WordNet, por ejemplo, es un recurso de este tipo en inglés que permite obtener sinónimos y antónimos:

*e.g. synonym sets containing “good”:*

```
from nltk.corpus import wordnet as wn
for synset in wn.synsets("good"):
    print "(%s)" % synset.pos(),
    print ", ".join([l.name() for l in synset.lemmas()])
    
(adj) full, good
(adj) estimable, good, honorable, respectable
(adj) beneficial, good
(adj) good, just, upright
(adj) adept, expert, good, practiced,
proficient, skillful
(adj) dear, good, near
(adj) good, right, ripe
...
(adj) well, good
(adj) thoroughly, soundly, good
(n) good, goodness
(n) commodity, trade good, good
```

*e.g. hypernyms of “panda”:*

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(pandaclosure(hyper))

[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

Figura 75. Información disponible en WordNet.

Fuente: <http://web.stanford.edu/class/cs224n/>

Este tipo de recursos funciona bien pero tiene una serie de **inconvenientes**:

- ▶ Muchas veces los matices se pierden. Por ejemplo, «*honorable*» aparece como sinónimo de «*good*», pero en muchos contextos una palabra no podría cambiarse por la otra.
- ▶ El lenguaje está en constante cambio con nuevos significados de palabras, lo que hace que este tipo de recursos pierdan ciertas acepciones modernas.
- ▶ Requiere mucho trabajo manual de un gran número de personas para recopilar una base de datos tan grande.
- ▶ Es complicado obtener una medida numérica de similitud entre palabras.

## Representaciones discretas del lenguaje

Tradicionalmente, el texto se ha representado computacionalmente mediante el uso de **representaciones discretas**. Empezando con un vocabulario  $V$ , se representa el texto mediante un vector del tamaño del número de elementos de  $V$ , donde cada elemento del vector representa una palabra. Una representación común es utilizar *term frequency*, donde se asigna el número de veces que aparece cada palabra en ese vector.

**Ejemplo 1.** Para un vocabulario  $V = [\text{perro}, \text{ gato}, \text{ mi}, \text{ tu}, \text{ simpático}, \text{ ladrador}, \text{ es}, \text{ era}]$ , la representación de la frase «mi perro es ladrador» sería:

[1, 0, 1, 0, 0, 1, 1, 0]

Esto nos dice el número de veces que aparece cada palabra en el texto y nos da una representación discreta del mismo, donde cada palabra se asocia a un valor del vector. En particular, esta representación se conoce como ***bag-of-words***. En *bag-of-words* no nos interesa el orden de las palabras, solo su aparición o no en el texto (de ahí la idea de que metemos las palabras en una «bolsa»).

**Ejemplo 2.** Con esta representación, una palabra sería expresada mediante un vector sencillo. Por ejemplo, una palabra como «hotel» podría ser representada utilizando un vocabulario distinto al del ejemplo anterior, como [0, 0, 0, 0, 1, 0].

Este tipo de representación discreta **pierde una gran cantidad de información** sobre la palabra. Por ejemplo, la palabra «motel» guarda una gran relación con «hotel». Si estamos buscando hoteles por Internet, los resultados con «motel» también serían relevantes. Sin embargo, la representación discreta de «motel» sería un vector del tipo [1, 0, 0, 0, 0, 0], que es totalmente ortogonal al vector de «hotel», haciendo imposible ver cualquier tipo de similitud entre ellos.

El objetivo sería pues, obtener una representación de una palabra que nos **aporte información semántica** sobre ella y que nos permita obtener un concepto de similitud entre representaciones de palabras. La idea será intentar codificar toda esa

información en un vector denso, llamado ***word vector***, lo que nos permitiría comparar palabras y utilizar esas representaciones en nuestros modelos de aprendizaje automático.

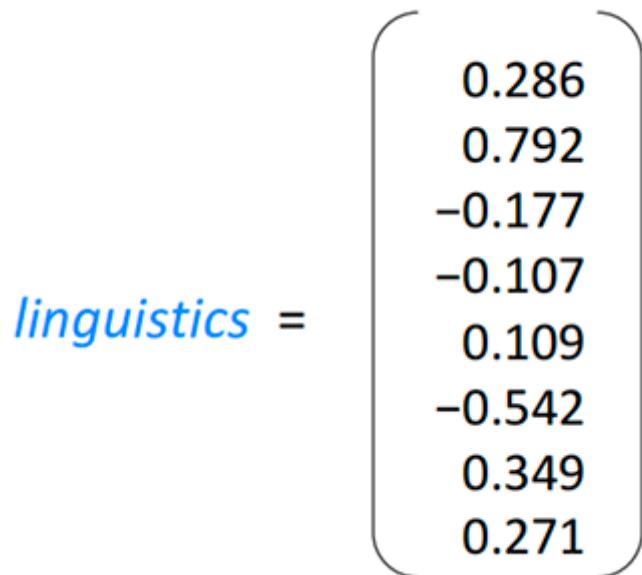


Figura 76. Representación distribuida de una palabra en un vector.

Fuente: <http://web.stanford.edu/class/cs224n/>

La representación de una palabra mediante un vector denso se conoce como  
**representación distribuida**.



Accede a los ejercicios de autoevaluación a través del aula virtual

### 6.3. Word2Vec



Accede al vídeo «Word2Vec» a través del aula virtual

Una vez que el objetivo que buscamos está claro, ¿cómo construimos un vector que represente el significado de una palabra? La idea clave que utilizaremos es que el significado de una palabra viene dado por las palabras que aparecen frecuentemente junto a ella.

Esta idea nos permite definir un procedimiento estadístico para construir el vector de una palabra. Si tenemos una gran cantidad de texto disponible, podemos extraer el significado de las palabras mediante el contexto en el que están presentes. Definiremos el **contexto de una palabra** como el conjunto de palabras que aparecen cerca de ella, dentro de un rango de tamaño determinado por nosotros, que definiremos como *window* o ventana.



...government debt problems turning into **banking** crises as happened in 2009...  
...saying that Europe needs unified **banking** regulation to replace the hodgepodge...  
...India has just given its **banking** system a shot in the arm...

Figura 77. Contexto de la palabra «*banking*».

Fuente <http://web.stanford.edu/class/cs224n/syllabus.html>

En la imagen anterior podemos ver el contexto de la palabra «*banking*» en varios textos. Estadísticamente, esta palabra aparecerá frecuentemente en contextos con palabras y conceptos parecidos. Sus sinónimos o palabras relacionadas aparecerán del mismo modo en el mismo tipo de contextos.

La idea aquí es obtener un vector denso para cada palabra de manera que sea similar a los vectores de palabras que aparecen en contextos similares, lo que indica que las palabras guardan una relación semántica. Estos vectores serán nuestros *word vectors*, también conocidos como *word embeddings* o *word representations*.

## Algoritmo word2vec

El algoritmo que utilizaremos para construir los *word vectors* es conocido como word2vec y fue presentado en 2013 (Mikolov, Chen, Corrado y Dean, 2013). El algoritmo se emplea sobre un corpus grande de texto (por ejemplo, todo el texto en un idioma de Wikipedia), sobre el que se obtiene un vocabulario de palabras disponibles. El *output* será un vector por cada palabra en el vocabulario.

El algoritmo funciona, a grandes rasgos, de la siguiente manera:

- ▶ Se recorre cada posición  $t$  en el texto, obteniendo una palabra central  $o$  y un contexto de palabras  $c$ .
- ▶ Utilizando las similitudes de los vectores calculados hasta el momento, se obtiene la probabilidad de  $c$  dado  $o$ .
- ▶ Se ajustan los vectores para maximizar esta probabilidad.

Por ejemplo, veamos las probabilidades que existen en una ventana (contexto) de tamaño 2:

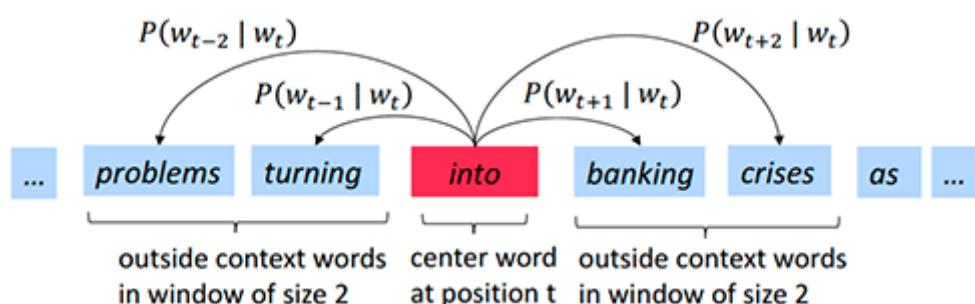


Figura 78. Ejemplo en una ventana de tamaño 2.

Fuente: <http://web.stanford.edu/class/cs224n/>

En la siguiente posición  $t$  en el texto tendremos:

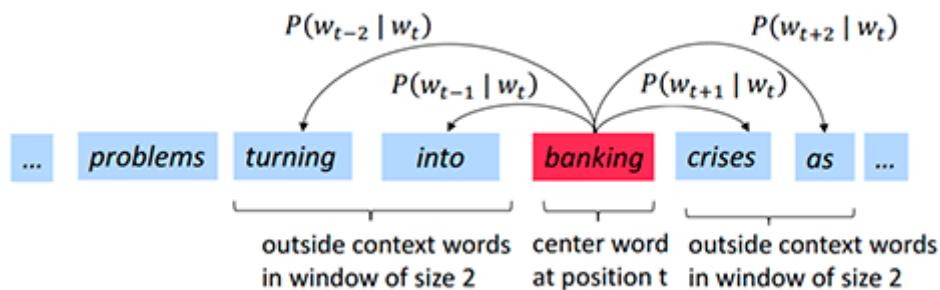


Figura 79. Ejemplo en una ventana de tamaño 2 en otra posición  $t$ .

Fuente: <http://web.stanford.edu/class/cs224n/>

De este modo, podemos obtener para todas las posiciones en el texto una verosimilitud o *likelihood* total a partir de las probabilidades de que una palabra esté

en el contexto dada una palabra central. Los parámetros de esta distribución de probabilidad serán todos los *word vectors* que queremos calcular:

$$L(\theta) = \prod_{t=1}^T \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j} | w_t; \theta)$$

Donde:

- ▶  $T$  es el número total de palabras en los textos;
- ▶  $t$  es la posición actual;
- ▶  $m$  es el tamaño de ventana elegido.

La idea es maximizar esta verosimilitud o *likelihood*. Si maximizamos estas probabilidades vistas en los datos mediante la elección de unos *word vectors* adecuados, seremos capaces de, a partir de una palabra central, predecir las palabras del contexto.

Para maximizar esta probabilidad, obtendremos, como ya hemos visto en otros problemas de *machine learning*, una **función de coste a minimizar**. Como queremos maximizar  $L$ , la función de coste a minimizar será el valor negativo de  $L$ .

Por otro lado, en vez de minimizar  $-L$  directamente, se aplica el logaritmo de  $L$ . Esto es un truco para evitar problemas numéricos en los cálculos, ya que las probabilidades son números pequeños y el producto de números pequeños se vuelve cada vez menor. Al aplicar el logaritmo, los productos se convierten en sumas.

De este modo, la función de coste a minimizar es:

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^T \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j} | w_t; \theta)$$

El único ingrediente que nos falta por ver es **cómo obtener la probabilidad** dentro de la suma. Esta se deriva a partir de la similitud de vectores. Durante el proceso de aprendizaje, se definen dos vectores  $v$  y  $u$  para cada palabra (en vez de uno):

- ▶  $v_w$  es el vector de la palabra  $w$  cuando esta actúa como **palabra central**;
- ▶  $u_w$  es el vector de la palabra  $w$  cuando esta actúa como **palabra de contexto**.

Con esto, la probabilidad a calcular con respecto a la palabra central  $c$  y la palabra de contexto  $o$  se modela de la siguiente manera:

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Esto quedaría así en el ejemplo visto en las figuras 4 y 5:

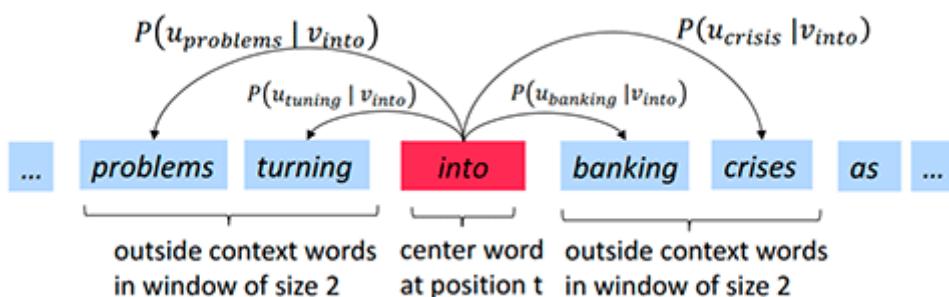


Figura 80. Probabilidad a calcular.  
Fuente: <http://web.stanford.edu/class/cs224n/>

## Similitud entre palabras (*word similarity*)

Para entender mejor cómo se obtiene esta probabilidad, veamos qué entendemos por similitud entre vectores. Para dos *word vectors*, una forma de obtener una medida de similitud es calcular su **producto escalar**. Este producto será mayor tanto para vectores que apuntan en la misma dirección como para valores absolutos grandes en direcciones similares.

La similitud mediante producto escalar puede hacerse arbitrariamente grande mediante valores mayores en los vectores, por eso es común utilizar el coseno del ángulo entre dos vectores como medida de similitud. El coseno de un ángulo tiene valores entre 0 y 1, y para dos vectores apuntando en la misma dirección, esto es, con ángulo 0°, tiene valor 1.

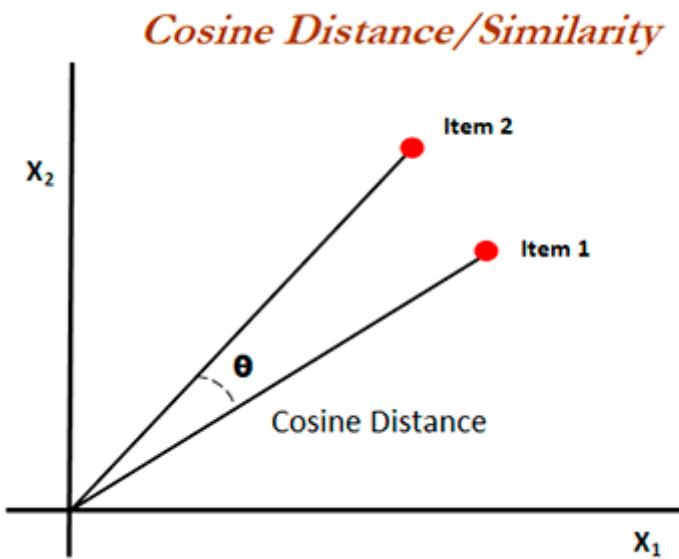


Figura 81. *Cosine Similarity*.

Fuente: <https://www.safaribooksonline.com/library/view/statistics-for-machine/9781788295758/eb9cd609-e44a-40a2-9c3a-f16fc4f5289a.xhtml>

La similitud por coseno o *cosine similarity* se calcula mediante la fórmula:

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \sum_{i=1}^n A_i B_i \sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}$$

Si bien lo común es utilizar *cosine similarity* como medida de *word similarity*, esto no es así en el caso de la probabilidad que estamos calculando en word2vec. En esta probabilidad se utiliza el producto escalar. La forma en la que el problema está dispuesto sobre un gran número de palabras impide, en cierta manera, que las similitudes se hagan arbitrariamente grandes mediante mayores valores en los vectores.

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

De este modo, volviendo a la fórmula podemos entender que la probabilidad de la palabra  $o$  dada la palabra central  $c$  es más grande cuanto más similares son.

## Función softmax

Igualmente, la función que se utiliza para modelar la probabilidad de una palabra en el contexto dada otra central no es casual. Si bien esta es una elección del modelo (se podría haber modelado la probabilidad de otro modo), se utiliza una función muy común en *machine learning* llamada *softmax*. Esta convierte cualquier serie de números en una distribución de probabilidad (haciendo que los valores de la distribución sumen 1).

Para un vector  $(x_1, \dots, x_n)$ , la función *softmax* es:

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)} = p_i$$

Como ejercicio para el alumno, se puede comprobar que los  $p_i$  suman 1.

El nombre de la función *softmax* viene de que se parece en cierta medida a un máximo, ya que la exponencial del numerador, el valor más grande del *input*  $(x_1, \dots, x_n)$  tiende a amplificarse. Sin embargo, como el resto de  $x_i$  aún tienen cierta probabilidad de salida, el máximo es «*soft*».

## Calculando los *word vectors*

Tenemos ya todos los ingredientes para calcular los *word vectors*. Hemos definido una función de coste basada en unas probabilidades modelizadas por dos tipos de

vectores por cada palabra. Una vez llegados aquí, sabemos que la solución al problema es aplicar *gradient descent* o SGD para obtener los valores de nuestros parámetros. En esta ocasión, nuestros parámetros no son los pesos y *biases* de una red neuronal, sino todos los valores de los vectores  $u$  y  $v$ :

$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \vdots \\ v_{zebra} \\ u_{aardvark} \\ u_a \\ \vdots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$

Figura 82. Cálculo de los *word vectors*.

Fuente: <http://web.stanford.edu/class/cs224n/>

El número de parámetros es  $2dV$ , donde:

- ▶  $V$  es el número de palabras en nuestro vocabulario.
- ▶  $d$  es la dimensión de los vectores.
- ▶ El número de parámetros se multiplica por 2 debido a que tenemos los vectores  $u$  y  $v$  por cada palabra.

Una vez resuelto el problema de optimización, se hace la media de los vectores  $u$  y  $v$  para obtener el *word vector* definitivo de la palabra. Podríamos preguntarnos para qué es necesario entonces tener dos vectores por palabra. Esto es así porque el problema de optimización es más sencillo de resolver de este modo y, además, los resultados experimentales son mejores.

### *Negative sampling*

En la práctica, no se suele aplicar *gradient descent* o SGD directamente sobre el problema definido arriba, ya que el número de cálculos a realizar se complica ante el tamaño del vocabulario, que puede ser muy grande. La solución consiste en aplicar *negative sampling*, una técnica en la que se toman las palabras que aparecen juntas en el contexto y se eligen al azar un pequeño número de palabras que no. El objetivo es «acercar» los vectores de las palabras que aparecen juntas y «alejar» los vectores de las que no sin tener que recurrir a tener que hacer cuentas con todas las palabras del vocabulario.

Si bien los detalles no son importantes para esta clase, la **función de coste** a implementar sería:

$$J_t(\theta) = \log \sigma(u_o^T v_c) + \sum_{i=1}^k \mathbb{E}_{j \sim P(w)} [\log \sigma(-u_j^T v_c)]$$

Figura 83. Función coste a implementar.

Donde:

- ▶ El primer término son las palabras que aparecen juntas.
- ▶ El segundo es una suma (*expectation*) sobre valores negativos elegidos al azar entre el total de palabras que no aparecen en la ventana o *window*.

### *Skip-grams y CBOW models*

Existen dos versiones de word2vec según cómo se calculen las probabilidades:

***Skip-grams***: es el modelo que hemos utilizado en esta clase. La idea es predecir las palabras del contexto a partir de la palabra central. Modelamos probabilidades sobre las palabras del contexto dada una palabra central.

***Continuous Bag of Words (CBOW)***: la idea aquí es predecir la palabra central a partir de una «*bag*» de palabras contexto. Normalmente, los vectores de las

palabras contexto se suman y modelamos la probabilidad de la palabra central.

## Resultados

Hemos empezado este tema buscando una representación de las palabras en forma de vector, de manera que las características semánticas de la palabra estuvieran presentes y pudiéramos obtener medidas de similitud entre palabras. La idea de word2vec es buscar este significado en el contexto en el que las palabras suelen aparecer.

El objetivo es ver de manera estadística qué palabras aparecen normalmente juntas o en contextos similares para extraer ese concepto de similitud.

Los resultados en forma de *word vectors* funcionan tal y como se espera. Si aplicamos una reducción de dimensionalidad para representar los vectores que se obtienen en dos dimensiones, obtenemos un mapa donde se ve cómo palabras similares aparecen juntas en clusters. Por ejemplo, Nokia con Samsung en el centro o una serie de nombres en la esquina superior derecha.



Figura 84. Mapa de word vectors.

Fuente: <http://web.stanford.edu/class/cs224n/>

La representación mediante *word vectors* también permite hacer cierta aritmética de vectores muy interesante. La diferencia entre los vectores de palabras como «*king*» y «*queen*» tiende a ser la misma que la de vectores como «*man*» y «*woman*». Este tipo de relaciones es común en el espacio vectorial resultante.

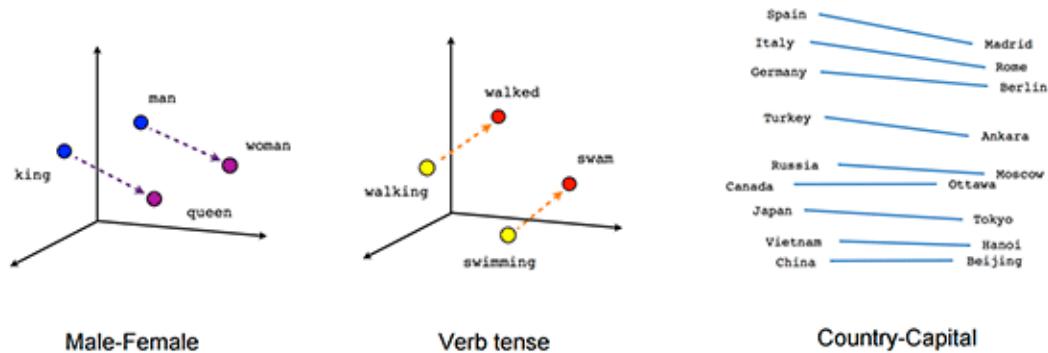


Figura 85. Ejemplos de representaciones de *word vectors*.

Fuente: <https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12>

Es importante mencionar, sin embargo, que esto no tiene por qué ocurrir siempre así. Al final, la calidad de los *word vectors* depende en gran medida de la **calidad y cantidad del texto** sobre el que se entrena. Para entrenarlos, se suelen obtener enormes cantidades de texto, tales como *dumps* completos de la Wikipedia o Google News.

Los *word vectors* son una pieza clave a la hora de entrenar modelos de *deep learning* para PLN. Por ello, existe una gran cantidad de vectores de palabras ya entrenados disponibles en Internet (*pre-trained vectors*). Estos se pueden utilizar directamente en nuestros modelos sin necesidad de entrenarlos nosotros mismos.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 6.4. Referencias bibliográficas

Mikolov, T., Chen, K., Corrado, G. y Dean, J. (2013). Efficient estimation of word representations in vector space. Recuperado de <https://arxiv.org/abs/1301.3781>



Accede al vídeo «Resumen» a través del aula virtual

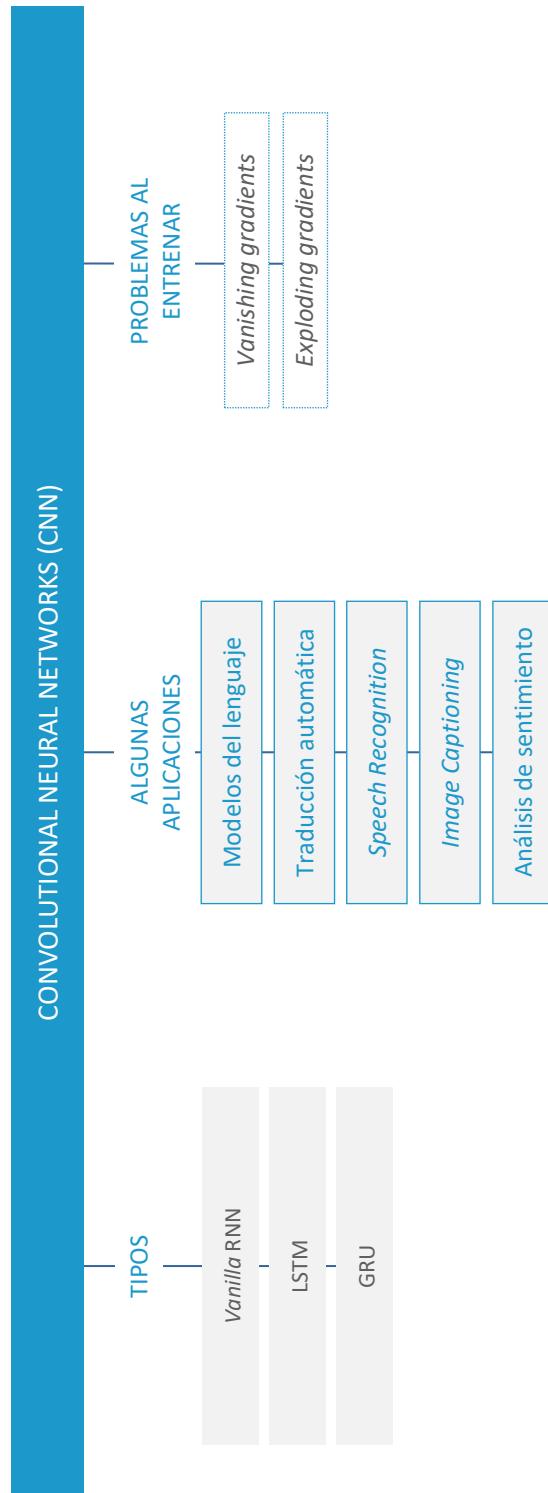
---

# Tema 7. *Recurrent Neural Networks (RNN)*

## Índice

Esquema	3
Ideas clave	4
7.1. Introducción y objetivos	4
7.2. <i>Recurrent Neural Networks</i>	4
7.3. Modelos del lenguaje con RNN	15
7.4. Arquitecturas LSTM y GRU	22

# Esquema



## 7.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

En este tema veremos otro de los tipos de redes neuronales que, junto a las CNN, ha revolucionado las aplicaciones de *machine learning* a varios problemas reales. Las *recurrent neural networks* o RNN tratan problemas sobre **secuencias** y están siendo fundamentales en áreas como el procesamiento del lenguaje natural.

Es importante comprender el funcionamiento de las *recurrent neural networks*, cómo se retroalimentan a lo largo del tiempo, cómo se utilizan durante una secuencia y cómo pueden producir una secuencia de salida. Al acabar el tema, hemos de ser capaces de relacionar ciertas aplicaciones con el uso de modelos recurrentes. También es fundamental comprender el caso particular de los modelos del lenguaje y de qué forma una RNN puede tratar el problema de manera natural, además de poder convertirse en un modelo capaz de generar lenguaje. Finalmente, es necesario conocer cuáles son los problemas que tienen las RNN sencillas y las arquitecturas más avanzadas que los solucionan.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 7.2. Recurrent Neural Networks



Accede al vídeo «*Recurrent Neural Networks*» a través del aula virtual

Hasta ahora hemos visto dos tipos de redes neuronales: *feed-forward neural networks* y *convolutional neural networks*. En ambas, tenemos un *input* de tamaño fijo (por ejemplo, una imagen) que produce una única salida (por ejemplo, una serie de probabilidades que permite determinar a qué clase corresponde la imagen). Sin embargo, en muchos problemas de *machine learning* nos gustaría tener más flexibilidad, con arquitecturas con una longitud variable de *inputs* o de *outputs*.

Como vimos en el tema anterior, un texto puede ser representado por un conjunto de palabras sin orden en lo que se conoce como *bag-of-words*. En este tipo de representación, todas las palabras del texto se juntan en un solo *input* discreto en forma de vector cuya dimensión es del tamaño del vocabulario; donde por cada palabra en el vocabulario se guarda el número de veces que esta palabra aparece en el texto. No obstante, algo que parece que tendría más sentido desde un punto de vista lingüístico sería representar ese texto como una secuencia ordenada de palabras, donde una red neuronal tomaría como *input* palabra a palabra hasta ver todo el texto.

Las *recurrent neural networks*, redes neuronales recurrentes en castellano o simplemente RNN, son un tipo de red neuronal que es capaz de trabajar con información secuencial, manteniendo un estado interno o **hidden state** (que podemos considerar como una memoria) para procesar secuencias de entrada. Este tipo de arquitectura aplica una fórmula recurrente sobre una secuencia de entrada de manera que, en cada paso dado en la secuencia, se depende del nuevo valor de *input*  $x$  y del estado interno anterior  $h$ . Por cada avance de la red en la secuencia, se suele decir que hemos avanzado un **time step**. Esto no quiere decir que todos los problemas que las RNN tratan sean temporales, si bien cualquier problema que puede desarrollarse en una serie temporal (por ejemplo, un vídeo, que es una secuencia de imágenes) es un gran candidato para ser tratado con una red recurrente.

La potencia de este tipo de redes radica en su capacidad de **modelar relaciones temporales entre elementos de la secuencia** a través del estado interno de la red,

que actúa como una suerte de memoria sobre lo que la red ha visto hasta ahora. No es, por tanto, sorprendente que este tipo de arquitecturas haya sido de gran importancia en problemas de procesamiento del lenguaje natural, donde el contexto y el orden son determinantes para la comprensión y tratamiento del texto.

## Ejemplos de problemas secuenciales

Antes de ver más formalmente el funcionamiento de las RNN, veamos una serie de problemas con secuencias que pueden ser naturalmente tratados con este tipo de redes neuronales.

El caso más sencillo podría ser el de una secuencia de entrada y una única salida. En este tipo de problema, tenemos una secuencia de entrada sobre la que aplicamos una red neuronal recurrente de la que tomamos una única salida al final de la secuencia.

**Ejemplo 1.** Podría ser el **análisis de sentimiento** en una oración o texto. El texto sería una secuencia de palabras sobre las que aplicaríamos la RNN y la salida sería la probabilidad de que el texto tenga un sentimiento asociado positivo o negativo.

En la imagen inferior podemos ver cómo se desarrollaría esto: cada rectángulo rojo es el *input* por cada palabra (por ejemplo, el *word vector* asociado a esta), los rectángulos verdes representan el *hidden state* de la red una vez que se aplica cada entrada y, finalmente, el rectángulo azul es la salida de la red una vez se ha desarrollado toda la secuencia.

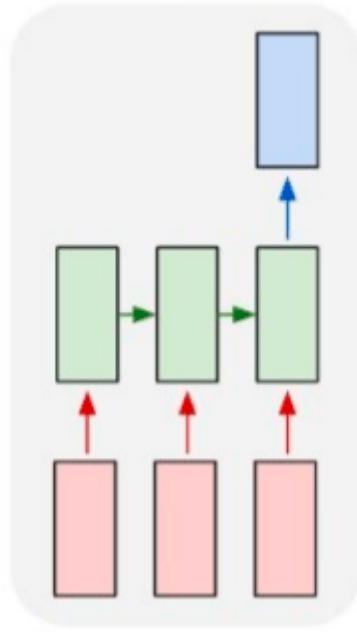


Figura 86. Secuencia de entrada de única salida.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

**Ejemplo 2.** Otro caso que puede tratarse con las RNN son los problemas de secuencia a secuencia. Por ejemplo, la **traducción automática** o *machine translation*, donde tenemos una secuencia de entrada (una frase) y la salida es otra frase en el idioma objetivo.

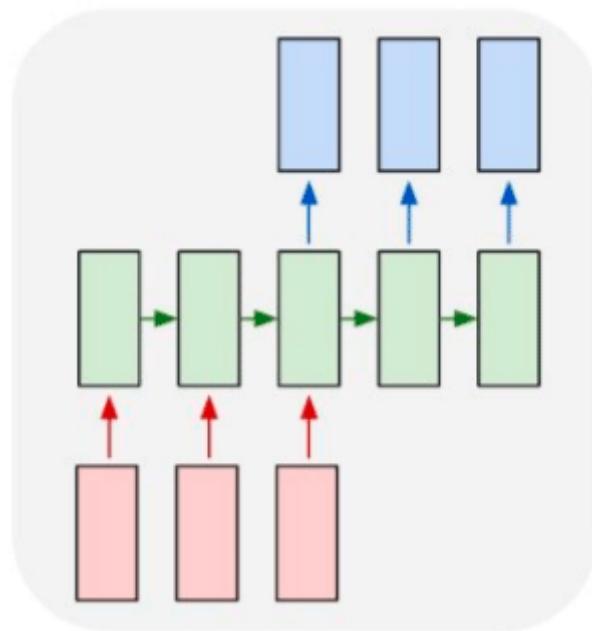


Figura 87. Secuencia de entrada con secuencia de salida.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

En este contexto solo empezamos a generar la secuencia de salida una vez que la secuencia de entrada se ha leído completamente, como sucede en problemas donde necesitamos tener toda la información de la secuencia de entrada antes de proceder con la salida, lo cual tiene sentido para el problema de traducción, ya que hay que «leer» el texto a traducir antes de lanzarse con la traducción. No obstante, esto no tiene por qué ser el caso en general, podemos igualmente generar una secuencia de salida a la vez que leemos la de entrada. Por ejemplo, en un vídeo podemos clasificar qué tipo de imagen hay en cada *frame*, a la vez que vamos leyendo las imágenes de entrada, se van clasificando las imágenes.

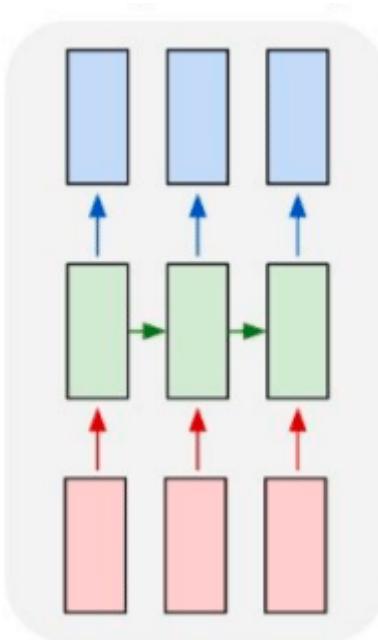


Figura 88. Otro tipo de secuencia de entrada con secuencia de salida.  
Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

**Ejemplo 3.** Incluso es posible generar una secuencia a partir de una sola entrada. Sería el caso del problema de *image captioning* que trata de extraer una pequeña frase que describa una imagen. Por ejemplo, «un niño jugando con juguete» a partir de la imagen del niño. Aquí obtenemos una secuencia de palabras a partir de la imagen de entrada.

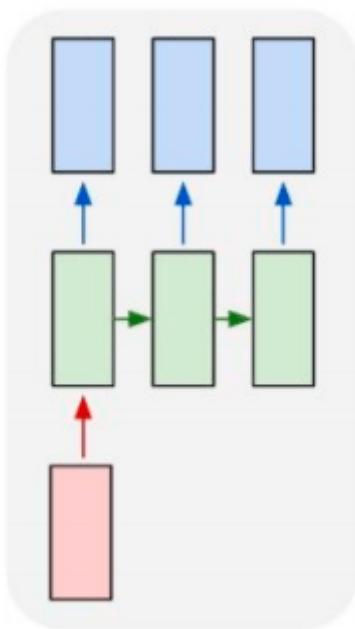


Figura 89. Secuencia a partir de una sola entrada.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

Nótese que esto es un esquema muy simplificado y que la arquitectura real de este problema es mucho más compleja e implica también CNN.

Como vemos, las RNN son muy dinámicas y pueden tratar una gran variedad de problemas de manera natural.

## Formulación de RNN

Como hemos dicho, una RNN es un tipo de red neuronal que toma una secuencia de valores de entrada y, de manera recurrente, aplica una transformación a partir de cada valor de entrada y del *hidden state* (estado interno) que posee la red en ese momento, obteniendo un nuevo estado interno y, de ser necesario, un nuevo valor de salida.

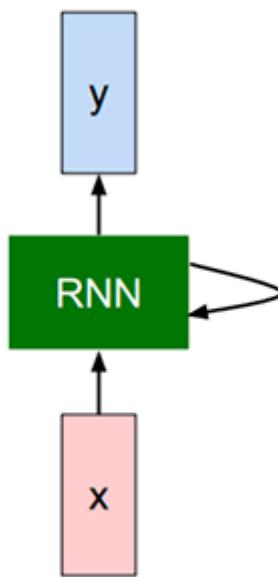


Figura 90. Representación esquemática de una RNN.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

De manera formal, la función de recurrencia de una RNN viene dada por:

$$h_t = f_w(h_{t-1}, x_t)$$

Donde:

- ▶  $h_t$  es el valor del *hidden state* (un vector) en el instante de tiempo  $t$ .
- ▶  $f_w$  es una transformación no lineal del estilo de las que hemos visto en otras redes neuronales, con  $w$  como una matriz de parámetros o *weights* de la red neuronal.
- ▶  $h_{t-1}$  es el valor del *hidden state* en el instante  $t - 1$ , esto es, el *hidden state* anterior de la RNN.
- ▶  $x_t$  es el valor de *input* para el tiempo  $t$  (de nuevo, un vector).

Como vemos, tanto  $x$  como  $h$  son vectores. La dimensión del vector interno  $h_t$  es un hiperparámetro de la red, mientras que la de  $x$  depende del problema que estamos tratando.

Esta fórmula es una fórmula general que viene a decir que el nuevo estado interno de la RNN depende del estado anterior y del *input* actual.

El tipo de RNN más estándar, a veces conocido como *vanilla* RNN, viene dado por la siguiente fórmula:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Donde:

- ▶  $W_{hh}$  y  $W_{xh}$  son las matrices de parámetros aplicadas a los vectores  $h_{t-1}$  y  $x_t$  respectivamente.

La fórmula es un producto matricial de una matriz por vector, de modo que el resultado final de hacer los productos y sumarlos es otro vector. Sobre este último se aplica la *nonlinearity*  $\tanh$  elemento a elemento, obteniendo el nuevo *hidden state*.

Finalmente, a partir del estado interno de la red recurrente podemos obtener la salida de la red  $y_t$  en el instante  $t$ :

$$y_t = W_{hy}h_t$$

Aquí estamos aplicando una transformación lineal sobre  $h_t$  mediante el producto de la matriz de pesos  $W_{hy}$  con el vector del *hidden state*. Nótese que durante esta clase no hemos visto la aplicación de transformaciones lineales en forma de productos matriciales. Este paso es equivalente al de aplicar una capa de una red neuronal estándar sin funciones de activación.

Con estas fórmulas tenemos el modo de avanzar una red recurrente sobre los valores de entrada y obtener una salida por cada *time step*  $t$  si el problema lo requiere. Es importante recalcar que los valores de los parámetros (en este caso, los valores de las matrices  $W_{hh}$ ,  $W_{xh}$  y  $W_{hy}$ ) no cambian en cada *time step*, es decir, un solo conjunto de parámetros es utilizado durante toda la secuencia.

## «Desenrollando» una RNN

La formulación como recurrencia de las RNN puede resultar un poco liosa. Sin embargo, es relativamente sencillo de visualizar si «desenrollamos» el grafo de computación para una red, tal y como se ve en la siguiente imagen:

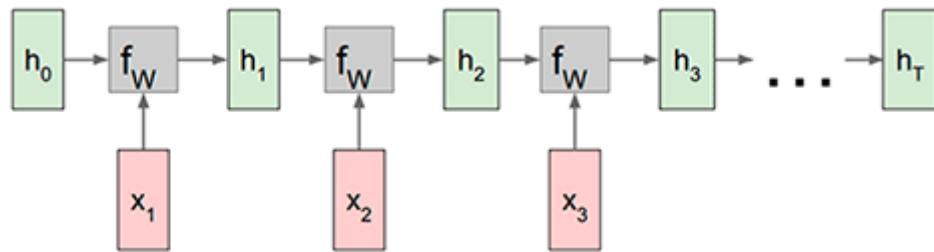


Figura 91. Grafo de computación de una RNN.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

Vemos cómo se aplica la función  $f_w$  al estado anterior de la red y a las nuevas *inputs* que llegan en cada *time step*, dando lugar a un nuevo estado. Igualmente, es posible obtener una salida por cada estado al que avanzamos, como se ve en la siguiente imagen. En ella también se recalca cómo los *weights* o pesos  $W$  tienen el mismo valor en todo el desarrollo de la secuencia.

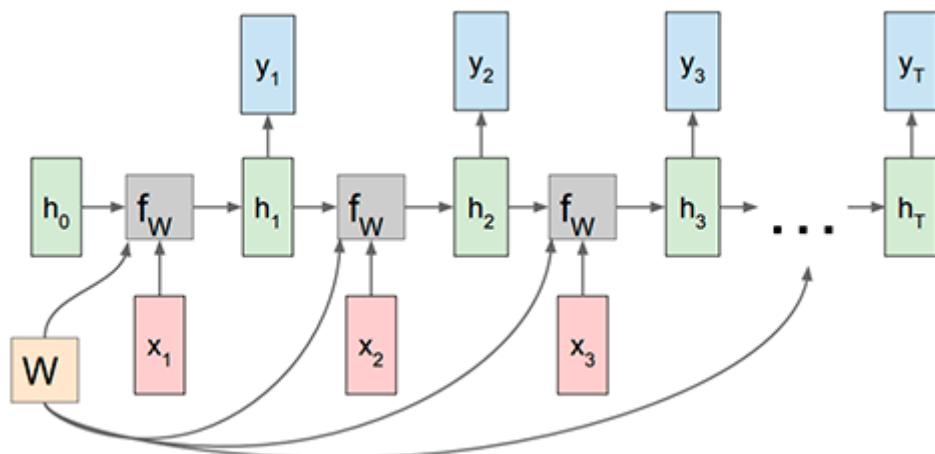


Figura 92. Grafo de computación de una RNN con *outputs*.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

Como vemos, es necesario también tener un estado inicial  $h_0$  de la red recurrente. Es muy común empezar con un vector de ceros.

De manera similar, podemos ver el caso de generar una secuencia con un solo *input*. En este caso, los vectores de *input* a partir de  $x_1$  desaparecen (son 0) y simplemente desarrollamos el estado interno de la *hidden state* durante varios *time steps*.

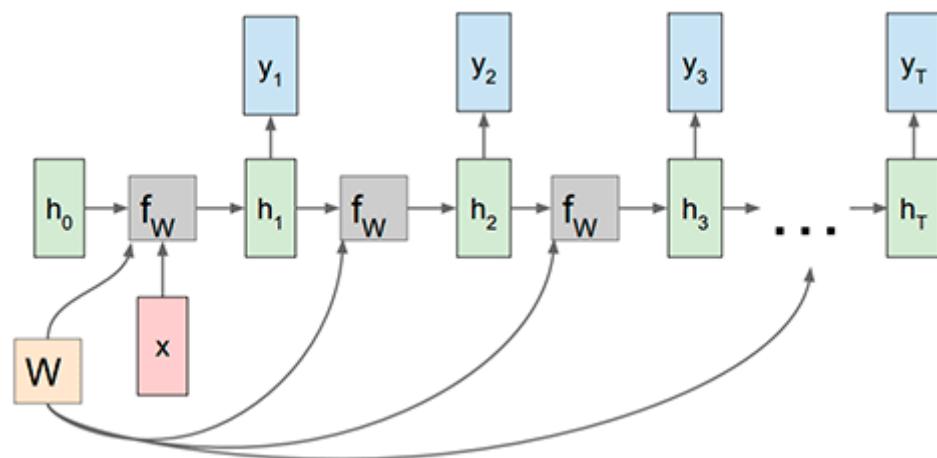


Figura 93. Secuencia a partir de un único *input*.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

### *Backpropagation through time*

Las RNN no son distintas al resto de redes neuronales en tanto en cuanto se utiliza el algoritmo de *backpropagation* para entrenarlas. En este caso, es necesario «desenrollar» la red a lo largo del tiempo y aplicar *backpropagation* a partir de todos los *outputs* que hemos obtenido (ver figura 9).

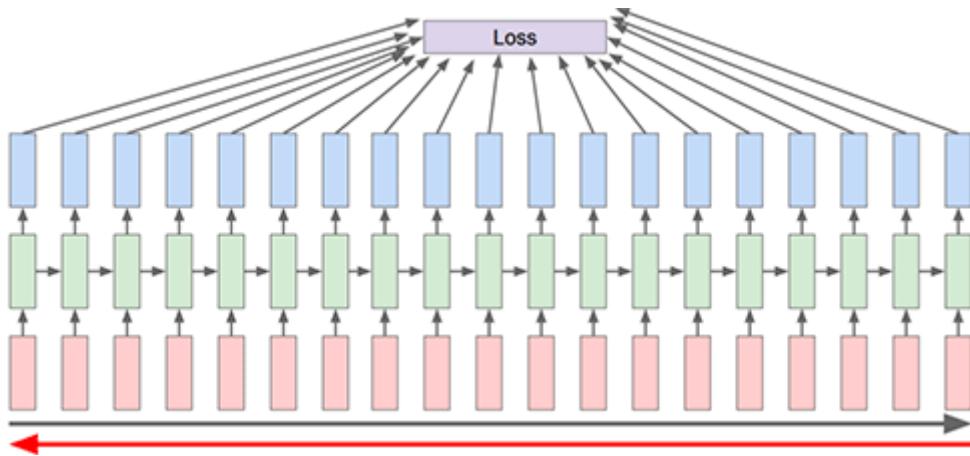


Figura 94. *Backpropagation Through Time* (BPTT).

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lec10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lec10.pdf)

Esto se conoce como ***backpropagation through time* (BPTT)**. La idea es hacer un *forward pass* a lo largo de toda la secuencia de entrada para calcular nuestra *loss* a partir de todos los *outputs*, y luego hacer el *backward pass* a lo largo de toda la secuencia. Si producimos una salida en cada *time step*, cada una de ellas tendrá cierta implicación en la *loss* calculada.

Este sistema puede hacerse computacionalmente muy costoso si nuestras secuencias de entrada son muy grandes. Por ello, en la práctica es normal utilizar ***truncated backpropagation through time* (TBPTT)**, donde la secuencia de entrada se «parte» en trozos y solo hacemos *backpropagation* en un pequeño número de *steps*. Por cada subsecuencia de entrada, el *hidden state* se mantiene desde la subsecuencia anterior, pero solo se hace *backpropagation* en la subsecuencia que estemos viendo en el momento (ver figura 10).

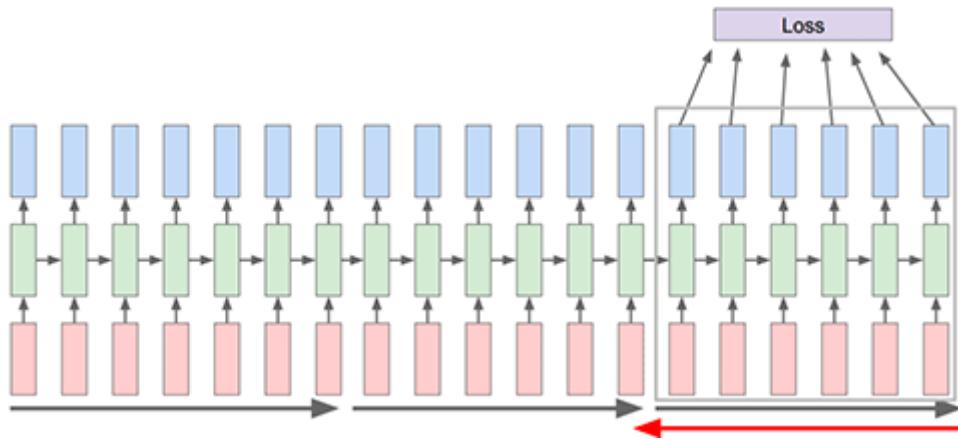


Figura 95. Truncated Backpropagation Through Time (TBPTT).  
 Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)



Accede a los ejercicios de autoevaluación a través del aula virtual

## 7.3. Modelos del lenguaje con RNN



Accede al vídeo «Modelos del lenguaje con RNN» a través del aula virtual

Veamos ahora una aplicación de las redes recurrentes donde estas han mejorado el estado de la técnica: los modelos del lenguaje, *language models* en inglés. Básicamente, un modelo del lenguaje es un modelo que asigna una probabilidad a una secuencia de palabras o incluso de caracteres:

$$p(w_1, w_2, \dots, w_T).$$

### Modelos del lenguaje tradicionales

Estos modelos son de gran utilidad. Por ejemplo, en traducción automática se suele tener una serie de traducciones candidatas, por lo que podemos valernos de la probabilidad que asigna el modelo a cada una de ellas para elegir la mejor traducción. Así, en castellano deberíamos de tener:

$$p(\text{"la casa es pequeña"}) > p(\text{"pequeña la casa es"})$$

Otro sitio donde podemos ver la aplicación de modelos del lenguaje es en la búsqueda de Google, donde se nos da una serie de opciones probables que la gente busca comúnmente:

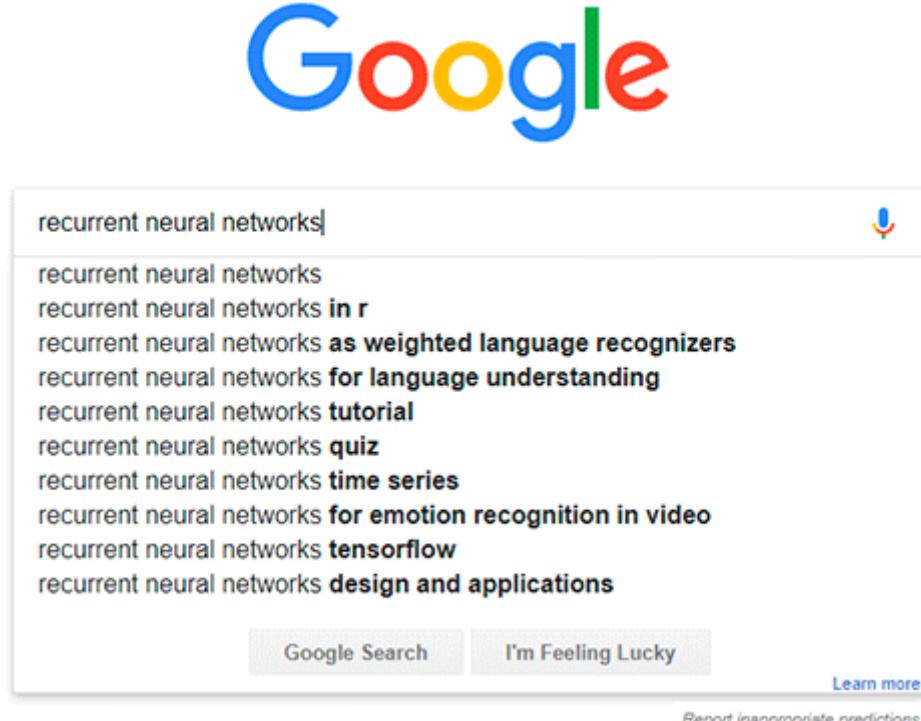


Figura 96. Ejemplo de modelo de lenguaje en el buscador de Google.

Fuente: [www.google.com](http://www.google.com)

Tradicionalmente, los modelos del lenguaje se han calculado de manera aproximada a través de probabilidades obtenidas mediante cuentas sencillas de palabras. La probabilidad de una serie de palabras para cada palabra viene dada por la probabilidad de que se dé esa palabra condicionada a las anteriores. Sin embargo, es normal aproximar esto con la suposición de que sería suficiente con conocer unas pocas palabras anteriores.

$$P(w_1, \dots, w_m) = \prod_{i=1}^m P(w_i | w_1, \dots, w_{i-1}) \approx \prod_{i=1}^m P(w_i | w_{i-(n-1)}, \dots, w_{i-1})$$

Estas probabilidades normalmente se estiman contando en los datos cuántas veces aparecen las palabras juntas. Por ejemplo, para calcular  $p(w_2|w_1)$ , es decir, la probabilidad de que la palabra  $w_2$  venga después de  $w_1$ , se cuentan todas las veces que las dos palabras aparecen en ese orden y se divide por el número de veces que aparece  $w_1$  en general.

$$p(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)}$$
$$p(w_3|w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2)}$$

Estos modelos sencillos, si bien funcionan de manera razonable, tienen un problema: si queremos condicionar una palabra a un número elevado de palabras anteriores, necesitaremos almacenar en memoria una cantidad enorme de cuentas con todas las posibles combinaciones de palabras. De este modo, estos modelos se vuelven computacionalmente intratables a mayor calidad de modelo deseada.

## Modelos del lenguaje con RNN

Como podemos ya imaginar, este problema puede tratarse de manera natural con *recurrent neural networks*. Es posible suministrar la secuencia de palabras a nuestra red neuronal paso a paso, intentando hacer que la red sea capaz de predecir la palabra siguiente. De este modo, el estado interno de la RNN codifica en cierta medida el texto visto hasta ahora, siendo por tanto capaz de condicionar la probabilidad de la siguiente palabra a lo visto anteriormente.

El *input* por cada *time step* puede ser un *word vector* preentrenado con las técnicas vistas en el capítulo anterior, o simplemente una representación *one-hot*, un vector con todo 0 menos un 1 en la posición de la palabra en el vocabulario. Como salida de la red en cada *time step*, tenemos una capa softmax que nos da la probabilidad de todas las palabras en el vocabulario a partir de la palabra vista en ese *time step* y del *hidden state* de la red, que sintetiza el resto de palabras vistas anteriormente. De este

modo, un modelo del lenguaje con RNN puede utilizarse como un **modelo generativo** capaz de generar textos a partir de un punto de inicio. Podemos pensar de nuevo en el Autosuggest de Google, que intenta predecir nuestra búsqueda mediante la generación de una serie de candidatos probables dentro del conjunto de búsquedas en el portal.

Veamos un ejemplo de un modelo del lenguaje a nivel de caracteres o letras. Curiosamente, no es necesario contar con modelos basados en palabras para ser capaces de modelar el lenguaje; los modelos basados en caracteres han demostrado una gran versatilidad y un funcionamiento muy adecuado al ser entrenados con RNN.

Para nuestro ejemplo, basado en el contenido que se puede ver en los apuntes del curso de CS231N (ver «Lo más recomendado»), tenemos un vocabulario de cuatro caracteres:

[h, e, l, o]

Por cada carácter, tendremos una representación *one-hot* en la entrada:

- ▶ «h» sería [1, 0, 0, 0].
- ▶ «l» sería [0, 0, 1, 0].

En la siguiente imagen podemos ver cómo sería el entrenamiento mediante la secuencia de entrada «hello»:

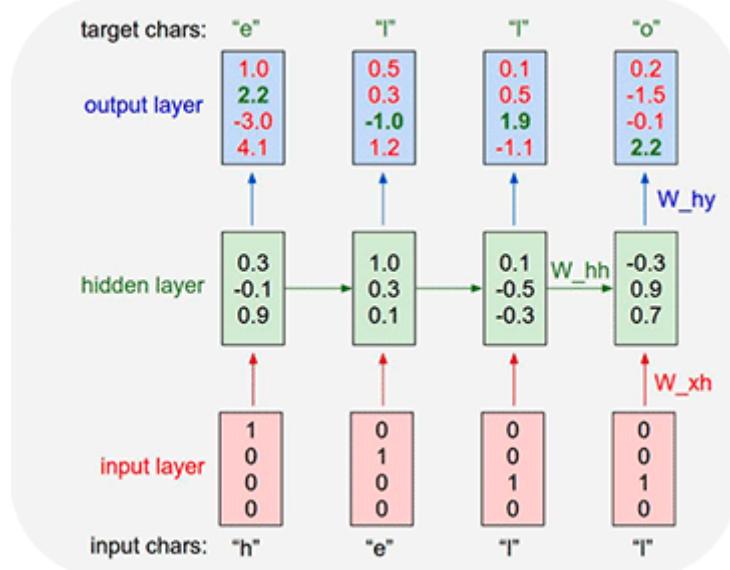


Figura 97. Entrenamiento de un modelo del lenguaje.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

En la imagen podemos ver cómo vamos introduciendo carácter a carácter, siendo la salida a predecir el siguiente carácter en la secuencia. En la *output layer* podemos ver los *scores* que la red neuronal predice por cada carácter (nótese que estos valores no son probabilidades, ya que no se ha pasado la función softmax). También se puede apreciar cómo el modelo se ha equivocado, por ejemplo, en el primer carácter predicho, ya que ha dado mayor puntuación a la «o» en vez de a la «e». Como sabemos, durante el entrenamiento el modelo se corregirá asignando una *loss* a ese fallo.

Podemos ver también dónde operan las distintas matrices de pesos  $W_{hh}$ ,  $W_{xh}$  y  $W_{hy}$  y cómo el *hidden state* se va actualizando con la nueva información que recibe.

Veamos ahora cómo podemos **utilizar nuestra RNN para generar texto**. Supongamos que la red ya ha sido entrenada y tenemos nuestras matrices  $W$  de pesos fijadas. Podemos hacer inferencia con la RNN para generar texto. Empezamos alimentando a la red con la letra «h»; al aplicar softmax a la última capa, la red nos da una serie de probabilidades para cada letra del vocabulario que pueden venir después de la «h». Utilizando estas probabilidades, hacemos un muestreo y obtener la siguiente letra a partir de la distribución de probabilidad que nos da la red. En el ejemplo, aunque la

letra más probable es la «o» con un 0.84 de probabilidad, al hacer el muestreo aleatorio hemos obtenido una «e». Esta letra obtenida se utiliza a su vez como entrada, con lo que obtendremos una nueva letra de salida, y así sucesivamente.

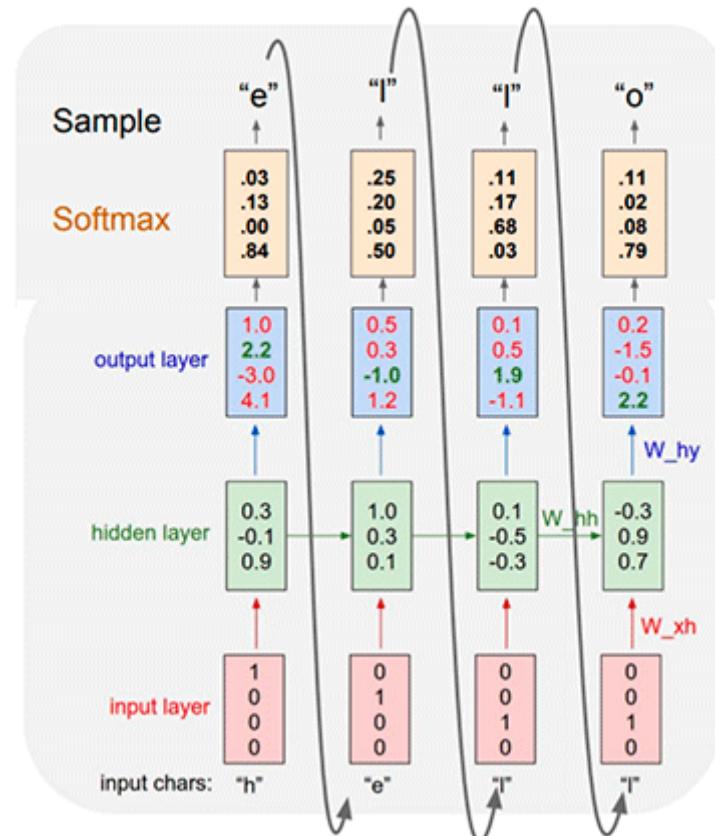


Figura 98. Uso de un modelo del lenguaje para generar texto.  
Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

Podríamos preguntarnos por qué aplicamos un muestreo sobre la distribución de probabilidad en vez de coger directamente la letra más probable. Si bien esto puede hacerse para obtener la combinación más probable, es normal utilizar estos modelos para generar distintas opciones y explorar el espacio de posibilidades, como se ve en el ejemplo del Autosuggest de Google.

Una demostración de lo que las redes recurrentes pueden ser capaces de hacer, veamos el resultado de entrenar un modelo del lenguaje basado en caracteres en las obras completas de Shakespeare:

PANDARUS:

Alas, I think he shall be come approached and the day  
When little strain would be attain'd into being never fed,  
And who is but a chain and subjects of his death,  
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,  
Breaking and strongly should be buried, when I perish  
The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.

Second Lord:

They would be ruled after this chamber, and  
my fair nues begun out of the fact, to be conveyed,  
Whose noble souls I'll have the heart of the wars.

Clown:

Come, sir, I will make did behold your worship.

VIOLA:

I'll drink it.

VIOLA:

Why, Salisbury must find his flesh and thought  
That which I am not aps, not a man and in fire,  
To show the reining of the raven and the wars  
To grace my hand reproach within, and not a fair are hand,  
That Caesar and my goodly father's world;  
When I was heaven of presence and our fleets,  
We spare with hours, but cut thy council I am great,  
Murdered and by thy master's ready there  
My power to give thee but so much as hell:  
Some service in the noble bondman here,  
Would show him to her wine.

KING LEAR:

O, if you were a feeble sight, the courtesy of your law,  
Your sight and several breath, will wear the gods  
With his heads, and my hands are wonder'd at the deeds,  
So drop upon your lordship's head, and your opinion  
Shall be against your honour.

Figura 99. Ejemplo de modelo del lenguaje entrenado en Shakespeare.

Fuente: Adaptado de [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

O, como otro ejemplo, un modelo del lenguaje entrenado en el código fuente en C del kernel de Linux:

```

static void do_command(struct seq_file *m, void *v)
{
    int column = 32 << (cmd[2] & 0x80);
    if (state)
        cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
    else
        seq = 1;
    for (i = 0; i < 16; i++) {
        if (k & (1 << 1))
            pipe = (in_use & UMXTHREAD_UNCCA) +
                ((count & 0x00000000fffff8) & 0x0000000f) << 8;
        if (count == 0)
            sub(pid, ppc_md.kexec_handle, 0x20000000);
        pipe_set_bytes(i, 0);
    }
    /* Free our user pages pointer to place camera if all dash */
    subsystem_info = &of_changes[PAGE_SIZE];
    rek_controls(offset, idx, &soffset);
    /* Now we want to deliberately put it to device */
    control_check_polarity(&context, val, 0);
    for (i = 0; i < COUNTER; i++)
        seq_puts(s, "policy ");
}

```

Figura 100. Modelo del lenguaje entrenado en el kernel de Linux.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

Como vemos, estos modelos tienen una gran capacidad de generar contenido con cierto sentido.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 7.4. Arquitecturas LSTM y GRU



Accede al vídeo «Arquitecturas LSTM y GRU» a través del aula virtual

Hasta ahora hemos visto redes recurrentes en su variante más sencilla, también conocida como *vanilla* RNN. Este tipo de RNN tiene una serie de problemas que ha

llevado a la creación de nuevas arquitecturas recurrentes más efectivas y fáciles de entrenar.

### *Vanishing gradients y exploding gradients*

Dos problemas muy típicos en el entrenamiento de redes neuronales recurrentes son los *exploding gradients* y los *vanishing gradients* (en castellano, algo así como «gradientes que explotan» y «gradientes que se desvanecen»). Estos problemas vienen del hecho de que tenemos que hacer *backpropagation* a lo largo de una secuencia larga de elementos.

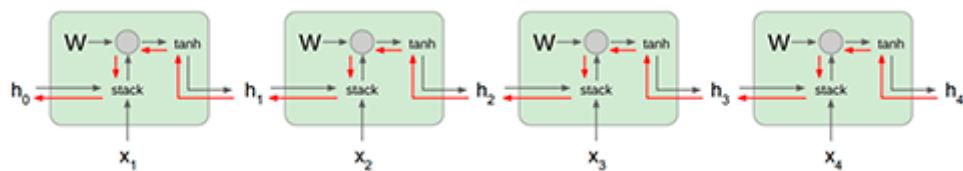


Figura 101. Grafo computacional de una secuencia con RNN.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

No entraremos en la formalización matemática de estos problemas, pero podemos intuirlo de manera sencilla. Imaginemos el gradiente que fluye hacia atrás hasta  $h_0$ . Este gradiente implica los mismos factores de  $W$  de manera continua, paso por paso (recordemos que los valores de  $W$  son los mismos a lo largo del tiempo). Pensemos en qué ocurre cuando multiplicamos un número por sí mismo en muchas ocasiones:

- ▶ Si este número es mayor que 1, obtenemos valores cada vez más grandes.
- ▶ Si este número está entre 0 y 1, vamos obteniendo valores más pequeños y cada vez más cercanos a 0.

Esto es lo que puede llegar a ocurrir en nuestras *vanilla* RNN. Según sean las propiedades de las matrices  $W$  y al hacer *backpropagation* en el tiempo, podemos encontrarnos con que:

- ▶ El gradiente pierda su valor de aproximación local y acabe divergiendo a un valor enorme (***exploding gradient***).
- ▶ O que se haga cada vez más pequeño hasta prácticamente desaparecer (***vanishing gradient***).

Como podemos imaginar, ambas situaciones afectan negativamente al entrenamiento de la red.

El problema de los *exploding gradients* puede solucionarse mediante un «truco» sencillo, conocido como ***gradient clipping***. La idea es medir la norma vectorial del gradiente en cada *time step* y, si esta supera cierto valor, dividir cada número del gradiente por una constante. Sin embargo, esto no es una solución óptima y, además, no ayuda en el problema de los *vanishing gradients*, por lo que se hacen necesarias mejores arquitecturas recurrentes.

## Arquitectura LSTM

La arquitectura LSTM (*Long Short-Term Memory*) data de 1997, por lo que podemos decir que sus creadores estaban muy adelantados a su tiempo, ya que la explosión en el uso de RNN no llegaría hasta más de quince años después. La fórmula de la arquitectura LSTM es un poco compleja y puede verse aquí comparada con una *vanilla* RNN:

### Vanilla RNN

$$h_t = \tanh \left( W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

### LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Figura 102. Comparación entre la arquitectura LSTM y una *vanilla* RNN.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

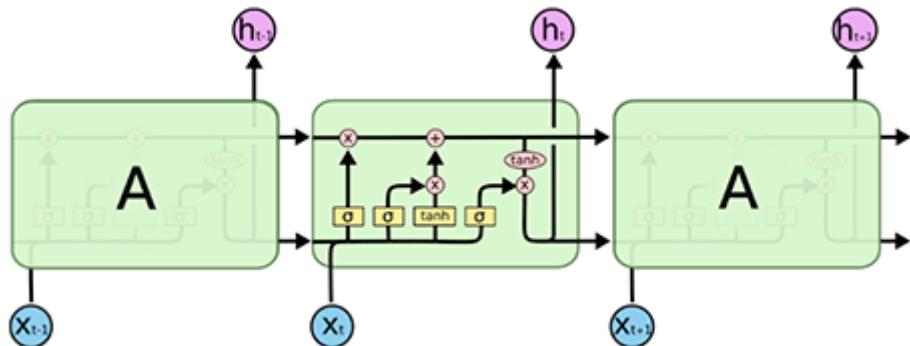


Figura 103. Arquitectura LSTM.

Fuente: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Las LSTM surgieron como una arquitectura encaminada a **solucionar los problemas de «memoria» de las *vanilla* RNN**. En la práctica, estas últimas presentan problemas para aprender relaciones con elementos de *time step* lejanos (es decir, que no están cerca del *time step* actual) debido a factores como los vistos en la sección anterior. Esto hace que gran parte del potencial teórico de las RNN se pierda. Por ejemplo, en el lenguaje es importante mantener información a lo largo de períodos relativamente largos, como podría ser mantener la información sobre el sujeto en la oración «Yo fui a casa después de ir al cine» cuando estamos analizando al final de la misma: «¿quién fue al cine?».

Las LSTM están diseñadas explícitamente para intentar solucionar este problema. Para ello, mantienen un estado interno *cell state* ( $c_t$ ) además del tradicional *hidden state* ( $h_t$ ), el cual representa una especie de «autopista de información» a lo largo del tiempo, como vemos en la siguiente imagen.

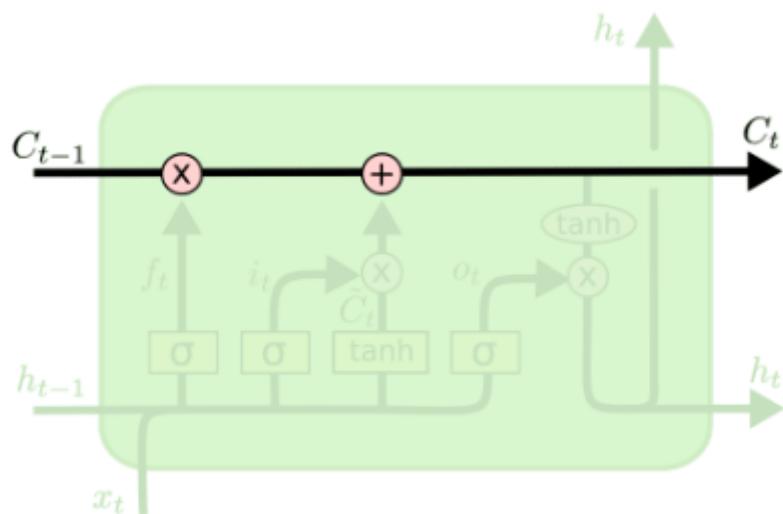


Figura 104. *Cell state* en la arquitectura LSTM.

Fuente: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Observando la fórmula de las LSTM, vemos que tenemos muchos elementos nuevos. En vez de calcular directamente « $h$ », ahora obtenemos cuatro vectores distintos conocidos como ***gates***:  $i, f, o$  y  $g$ , que después se combinan para obtener el *cell state*  $c$  y el *hidden state*  $h$ . Los productos que se ven entre vectores en la fórmula son productos elemento a elemento.

Como vemos,  $i, f$  y  $o$  resultan tras aplicar una función *sigmoid*, por lo que tienen valores entre 0 y 1. De este modo, actúan como *gates* que conservan (valores cercanos a 1) o eliminan (valores cercanos a 0) información. Más en particular:

- ▶  $f$  se conoce como ***forget gate*** y, al ser multiplicada por el *cell state* anterior ( $c_{t-1}$ ), representa cuánto tenemos que olvidar de los valores almacenados en este.
- ▶  $i$  se conoce como ***input gate*** y representa cuánto hemos de escribir en  $c_t$ . Los valores a escribir en  $c$  vienen dados por  $g$ , que viene de aplicar una *tanh* como en las RNN tradicionales.
- ▶  $c_t$  se obtiene de la mezcla de:
  - «Cuánto queremos recordar del pasado», el producto de  $f$  por  $c_{t-1}$ .
  - Y de «cuánta información nueva queremos añadir», el producto de  $i$  por  $g$ .

- ▶ Finalmente, el *hidden state* resultante se obtiene aplicando la transformación *tanh* sobre nuestro *cell state* y multiplicando por *o* (***output gate***), que nos dice cuánto de nuestro estado interno *c* hemos de revelar en el *hidden state*.

El funcionamiento de una LSTM es un poco lioso y puede parecer más magia que otra cosa. Sin embargo, su excelente rendimiento en la práctica ha sido fundamental para la popularidad de las RNN en el mundo del *machine learning*.

Finalmente, el hecho de que las LSTM no sufran de problemas de *vanishing* y *exploding gradients* viene de que los estados internos  $c_t$  crean una especie de «autopista de la información». Si bien no lo veremos formalmente aquí, el hecho de que  $c$  dependa de una interacción de sumas en vez de depender directamente de los parámetros  $W$  hace que se evite esa especie de multiplicación continua por el mismo valor, permitiendo que el flujo de gradientes hacia atrás sea más estable.

## Arquitectura GRU

Otra arquitectura que se ha hecho popular en la actualidad es la ***Gated Recurrent Unit*** (GRU). Fue introducida en 2014 y utiliza un sistema similar de *gates* al visto en la LSTM. Las mayores diferencias con LSTM son que se combina el *cell state* y el *hidden state* en un solo elemento, así como la *forget gate* y la *input gate* en una sola puerta.

$$\begin{aligned} r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\ z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\ \tilde{h}_t &= \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \end{aligned}$$

Figura 105. Fórmula de GRU.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

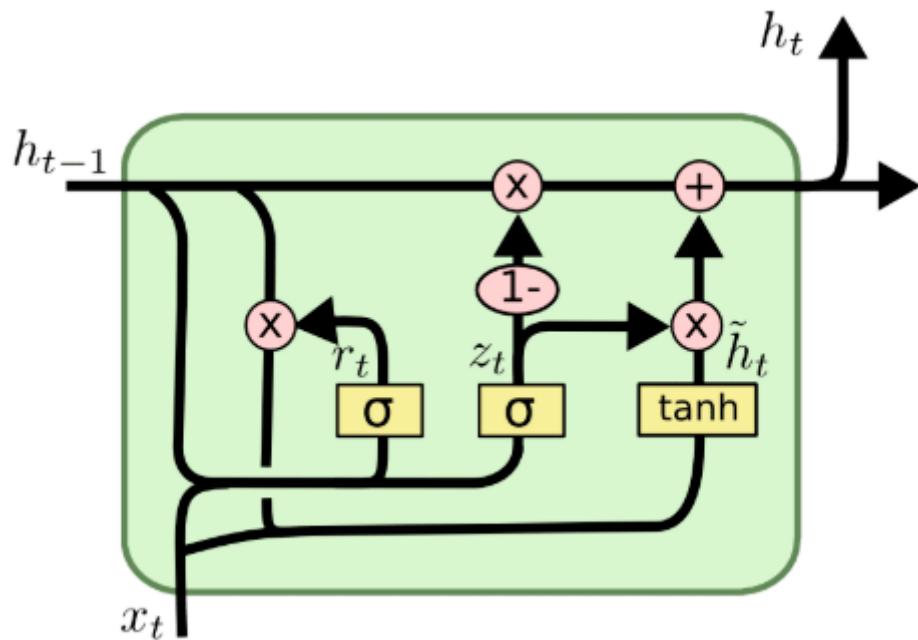


Figura 106. GRU.

Fuente: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

## Consideraciones prácticas

En la práctica, a la hora de abordar un problema con secuencias, lo más común es utilizar directamente LSTM o GRU. Estas se comportan mejor que las *vanilla* RNN en casi todas las situaciones, al no tener el problema de los *exploding* o *vanishing gradients* y ser capaces, por tanto, de modelar relaciones temporales más largas. La investigación en encontrar arquitecturas más efectivas o simples es un área muy activa, así como es el intentar comprender mejor cómo funcionan y aprenden estas redes desde un punto de vista más teórico.

Por otro lado, en la práctica suelen verse **arquitecturas profundas de RNN apiladas** (*stacked RNN*). En estas arquitecturas, la salida de una RNN se utiliza como la entrada secuencial de la RNN apilada encima de ella.

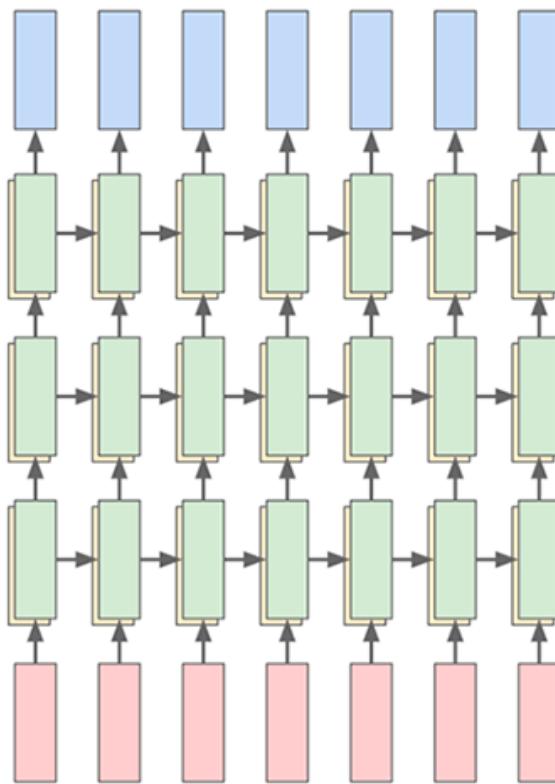


Figura 107. RNN apiladas.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture10.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf)

No es habitual apilar un gran número de LSTM o GRU, las profundidades más frecuentes son entre 3 y 5 RNN. Nótese que las RNN son bastante complejas computacionalmente y su entrenamiento requiere de bastante tiempo.

Finalmente, es también común ver **RNN bidireccionales** (*bidirectional RNN*), donde las secuencias de entrada se leen de atrás hacia delante y de delante hacia atrás.



Accede a los ejercicios de autoevaluación a través del aula virtual



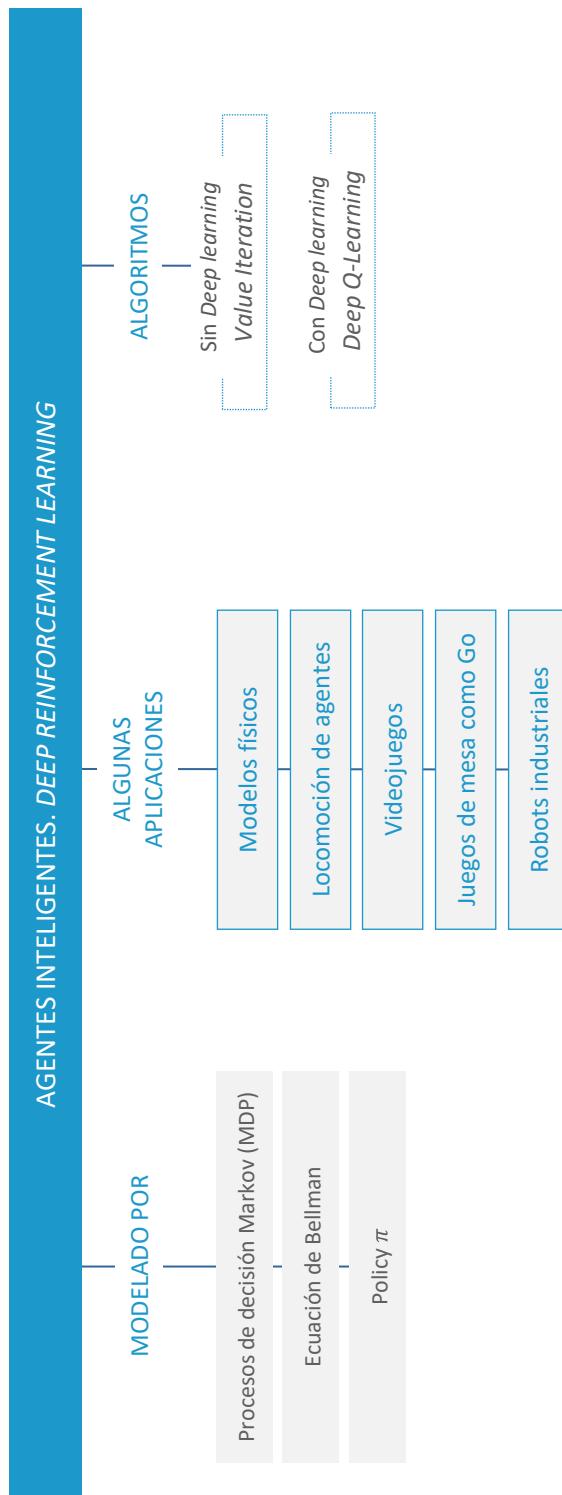
Accede al vídeo «Resumen» a través del aula virtual

# Tema 8. Agentes inteligentes. *Deep Reinforcement Learning*

## Índice

Esquema	3
Ideas clave	4
8.1. Introducción y objetivos	4
8.2. <i>Reinforcement Learning</i>	4
8.3. Procesos de decisión de Markov	9
8.4. <i>Deep Q-Learning</i>	15

# Esquema



## 8.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

En este tema veremos otra área que ha sido revolucionada por el aprendizaje profundo. La combinación de *deep learning* y *reinforcement learning* ha dado lugar a nuevos hitos en el mundo de la inteligencia artificial, con agentes capaces de aprender a jugar a juegos viendo directamente imágenes de las partidas o ganando a los humanos en juegos como el Go, considerados hasta hace poco algo fuera del alcance de una IA.

Es importante comprender qué es un problema de aprendizaje por refuerzo, cómo se diferencia de los problemas de aprendizaje supervisado vistos hasta ahora y cómo se define de manera formal mediante un proceso de decisión de Markov. Igualmente, es necesario aprender cómo podemos combinar esto con las redes neuronales profundas para dar lugar al *Deep Q-Learning*, así como entender el funcionamiento de este algoritmo.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 8.2. Reinforcement Learning



Accede al vídeo «Reinforcement Learning» a través del aula virtual

Hasta ahora hemos estado trabajando principalmente en problemas de **aprendizaje supervisado**. En este tipo de *machine learning*, tenemos una serie de datos  $x$  y una

salida  $y$  que queremos predecir. El objetivo es aprender una función capaz de llevarnos de  $x$  a  $y$ . En nuestro caso, esta función ha venido dada por redes neuronales. El proceso de entrenamiento ha consistido en utilizar los pares  $(x, y)$  conocidos para intentar aprender esa función. Un ejemplo de este tipo de problema es la clasificación de objetos en imágenes, con  $x$  siendo la imagen e  $y$  siendo el objeto a predecir.

Otro tipo de aprendizaje automático es el **aprendizaje no supervisado**; aquí tenemos solo los datos  $x$ , sin valor de salida a predecir. El objetivo es aprender una estructura existente en los datos. Problemas de este tipo incluyen *clustering*, *density estimation*, reducción de dimensionalidad, etc.

En este tema, vamos a tratar un tipo distinto de aprendizaje, el **aprendizaje por refuerzo** o *reinforcement learning*. En este tenemos:

- ▶ Un agente interactuando con un entorno (*environment*).
- ▶ El agente se encuentra en un estado  $s$  y lleva a cabo acciones  $a$ .
- ▶ Lo cual produce un nuevo estado y una recompensa (*reward*) del entorno.

El objetivo aquí es aprender las acciones a realizar según nuestro estado para maximizar la recompensa.

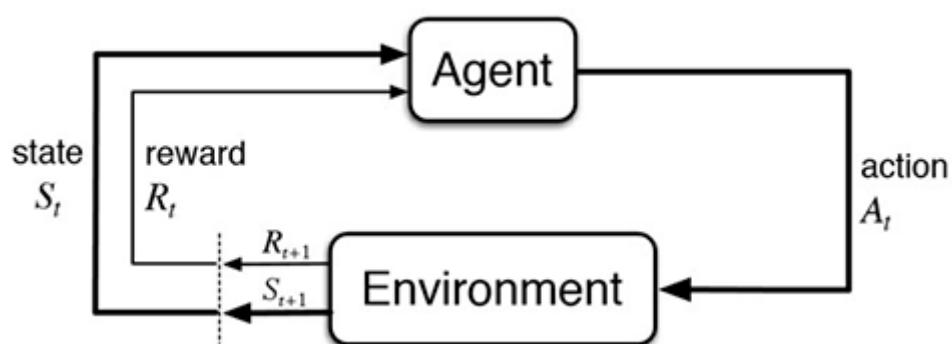


Figura 108. Aprendizaje por refuerzo o *reinforcement learning*.

Fuente: <https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html>

## Ejemplos

**Ejemplo 1.** El problema del *cart-pole* o péndulo invertido: tenemos un poste con la forma de un péndulo invertido encima de un carro en movimiento. El objetivo es mover el carro de modo que el poste no caiga y se mantenga en posición vertical.

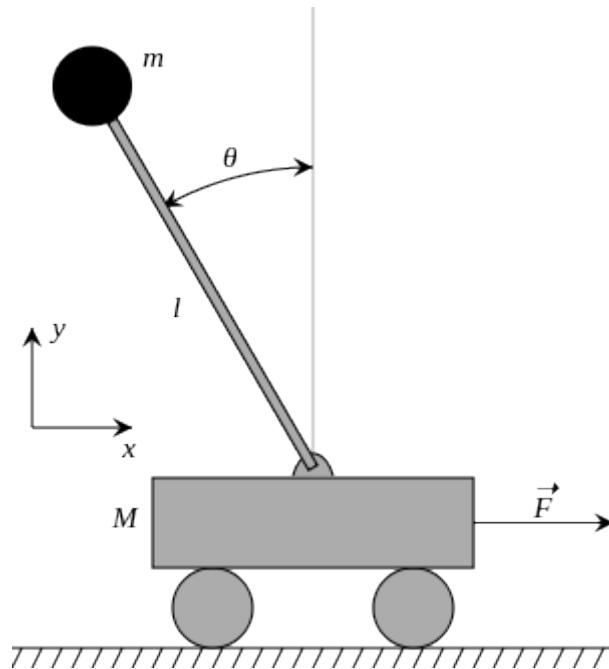


Figura 109. Dibujo esquemático de un péndulo invertido sobre un carro.

Fuente: [https://en.wikipedia.org/wiki/Inverted\\_pendulum](https://en.wikipedia.org/wiki/Inverted_pendulum)

En este problema:

- ▶ El **estado** en el entorno viene dado por:
  - El ángulo en el que se encuentra el poste.
  - La velocidad angular que lleva en ese momento.
  - La posición del carro.
  - La velocidad del carro.
- ▶ El **agente** puede realizar acciones como aplicar una fuerza al carro en dirección horizontal.
- ▶ La **recompensa** o *reward* es 1 por cada instante de tiempo en el que el péndulo esté en estado vertical. De este modo, el agente deberá aprender a, según un estado inicial, mover el carro hacia los lados de modo que el péndulo se estabilice.

**Ejemplo 2.** El movimiento de robots (*robot locomotion*). En este tipo de problemas, se define un entorno físico por medio de un simulador en el que unos robots-agentes con distintas formas (humanoides o arañas, por ejemplo) tienen que aprender a moverse hacia delante manteniéndose en equilibrio y esquivando ciertos obstáculos.

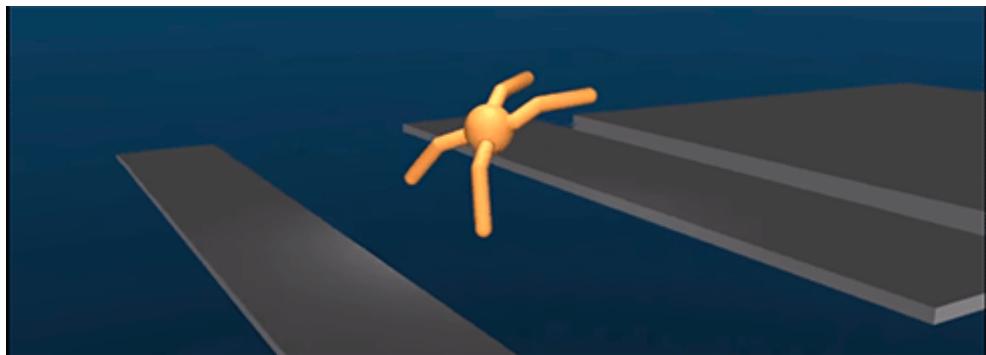


Figura 110. Robot araña aprendiendo a saltar.

Fuente: [https://en.wikipedia.org/wiki/Inverted\\_pendulum](https://en.wikipedia.org/wiki/Inverted_pendulum)

En este caso:

- ▶ El estado del entorno viene dado por una posición en el plano físico y por la posición y movimiento de cada una de las articulaciones que definen al agente.
- ▶ Las acciones se corresponden con movimientos posibles de estas articulaciones para moverse.
- ▶ El agente es recompensado por moverse hacia delante. De este modo, este aprende a utilizar sus articulaciones para caminar y esquivar obstáculos.

**Ejemplo 3.** Otro ejemplo de *reinforcement learning* aplicado a tareas reales está en los videojuegos. Un agente inteligente puede aprender a jugar a videojuegos clásicos como Doom directamente a partir de las imágenes del juego. Esto es radicalmente diferente a las IA que normalmente vemos programadas en los videojuegos.

En este caso, podemos definir un problema de aprendizaje por refuerzo donde:

- ▶ El estado viene dado por lo que un jugador ve.
- ▶ Las acciones son el movimiento en todas las direcciones y disparar
- ▶ El agente es recompensado cuando elimina a un oponente y es castigado (recompensa negativa) cuando muere.



Figura 111. Imagen del videojuego Doom.

Fuente: <https://itunes.apple.com/us/app/doom-classic/id336347946?mt=8>

**Ejemplo 4.** Finalmente, un ejemplo más orientado a la industria sería un robot para cadenas de montaje como podría ser un robot encargado de meter objetos en una caja.



Figura 112. Robot de una cadena de montaje.

Fuente: <https://youtu.be/MQ6pP65o7OM>



Accede a los ejercicios de autoevaluación a través del aula virtual

## 8.3. Procesos de decisión de Markov



Accede al vídeo «Procesos de decisión de Markov» a través del aula virtual

Los procesos de decisión de Markov, *Markov Decision Processes* (MDP) en inglés, son la formulación matemática de los problemas de aprendizaje por refuerzo o *reinforcement learning*. Estos procesos se definen mediante una tupla:

$$(S, A, P_{sa}, \gamma, R)$$

Donde:

- ▶  $S$  es el conjunto de **estados**.
- ▶  $A$  es el conjunto de **acciones**.
- ▶  $P_{sa}$  son las probabilidades de **transiciones** entre estados. Para cada estado  $s$  y acción  $a$ ,  $P_{sa}$  es una distribución de probabilidad sobre los posibles estados destino. Básicamente, nos da las probabilidades de a qué estado iremos si tomamos la acción  $a$  en el estado  $s$ .
- ▶  $\gamma$  es el **discount factor**. Es un número entre 0 y 1 y representa la importancia de obtener *rewards* lo más pronto posible.
- ▶  $R$  es la **reward function**, y modela las recompensas dadas a partir de un estado y acción.

Este modelo matemático es suficientemente genérico como para permitir transiciones probabilísticas entre estados, definidas por  $P_{sa}$ . En muchos problemas, sin embargo, las transiciones son deterministas y se corresponden con unas probabilidades de 1 por cada estado y acción. Podemos pensar que este es el caso si resulta complicado pensar en un mundo de probabilidades.

Los procesos de Markov siguen la **propiedad de Markov**, que dice que el estado actual caracteriza completamente el estado del entorno. Es decir, a la hora de realizar una acción  $a_t$  en un estado  $s_t$ , la transición al nuevo estado al que vamos es independiente de los estados anteriores en los que podía encontrarse el mundo.

Un proceso de Markov funciona de la siguiente manera. En el *time step* inicial ( $t_0$ ) empezamos en un estado inicial ( $s_0$ ). A partir de ahí, mientras no lleguemos a un estado considerado terminal:

5. El agente selecciona una acción  $a_t$ .
6. El entorno otorga una recompensa  $r_t$  basada en el estado actual  $s_t$  y la acción  $a_t$ .
7. Según la distribución de probabilidad  $P_{sa}$ , el entorno obtiene un nuevo estado  $s_{t+1}$  para el agente.

El agente elegirá acciones con base en una política  $\pi$  (**policy  $\pi$** ).  $\pi$  es por tanto una función que va de los estados  $S$  a las acciones  $A$  y determina las acciones que el agente tomará en cada estado.

**El objetivo es encontrar la política óptima  $\pi^*$  que maximice la suma  
cumulativa de recompensas a lo largo del tiempo.**

- ▶ Estas recompensas vienen dadas por  $r_t$ .
- ▶ Hemos definido también el *discount factor*  $\gamma$ , que define un descuento del valor de las recompensas a lo largo del tiempo.
- ▶ Formalmente, queremos encontrar la política que maximice la suma cumulativa descontada de las recompensas:

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | \pi \right]$$

Figura 113. Fórmula de la política óptima.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

Como tenemos ciertos valores aleatorios, ya que las transiciones  $P_{sa}$  han sido definidas de manera estocástica y lo mismo podría hacerse con el estado inicial y las *rewards*, la fórmula toma una esperanza matemática. Como vemos, el *discount factor*  $\gamma$  se multiplica por sí mismo en cada *time step*. Por tanto, tenemos que:

- ▶ Estas recompensas vienen dadas por  $r_t$ .
- ▶ Si  $\gamma = 1$ , no hay descuento.
- ▶ Cuanto más pequeño sea  $\gamma$ , menos sumarán las recompensas futuras en la búsqueda de la política óptima (*optimal policy*).

Veamos un ejemplo de *optimal policy* en un problema sencillo. Imaginemos que tenemos la siguiente cuadrícula, donde cada cuadro es un estado y las posibles acciones son moverse en las cuatro direcciones.

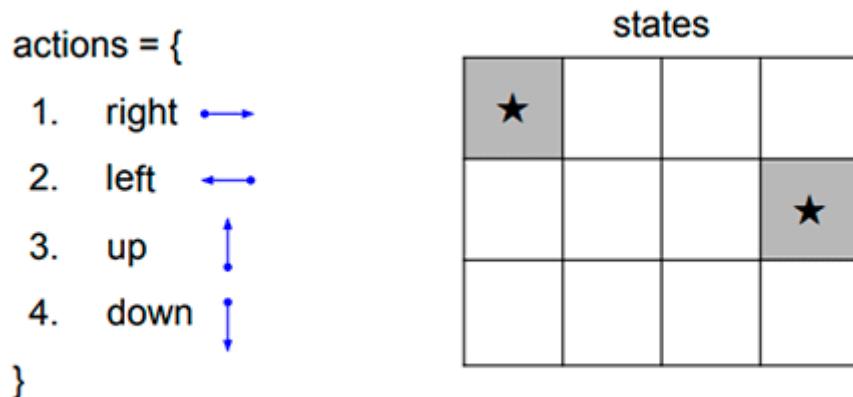
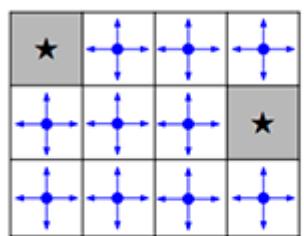


Figura 114. Ejemplo de MDP para encontrar la *optimal policy*.

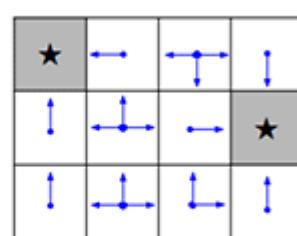
Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

El objetivo es alcanzar uno de los estados marcados con una estrellita en el **menor número de acciones**. Por ello, a cada movimiento en la cuadrícula se le asigna una recompensa negativa (por ejemplo,  $r = -1$ ). De este modo, el agente está forzado a llegar a un estado terminal cuanto antes para evitar acumular valores negativos de recompensas.

Con esta información, podemos representar la *optimal policy* de la siguiente manera y compararla con una *policy* aleatoria:



Random Policy



Optimal Policy

Figura 115. Comparación entre *optimal policy* y una aleatoria.  
Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

### Value function y Q-value function

Seguir una *policy* produce una trayectoria a lo largo del problema. Tal y como hemos visto, en esta estamos en un estado, realizamos una acción y recibimos una recompensa y un nuevo estado.

$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2 \dots$$

Vamos a definir ahora dos conceptos fundamentales en *reinforcement learning* que nos permiten evaluar cómo de bueno es un estado y una acción a partir de la trayectoria potencial que podemos tomar a partir de estos.

La **value function**:  $V$  en el estado  $s$  para una *policy*  $\pi$  es la recompensa acumulada esperada a partir de seguir la *policy* desde el estado  $s$ :

$$V^\pi(s) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi \right]$$

Figura 116. *Value function*.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

La ***Q*-value function**:  $Q$  en el estado  $s$  y acción  $a$  para una *policy*  $\pi$  es la recompensa acumulada esperada al elegir la acción  $a$  en el estado  $s$  y, después, seguir la *policy*:

$$Q^\pi(s, a) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Figura 117. *Q-value function*.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

Con esto, la **optimal Q-value function**  $Q^*$  es el máximo valor alcanzable con una *policy*  $\pi$  a partir de un par (estado, acción):

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Figura 118. *Optimal Q-value function*.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

Conociendo  $Q^*$ , la **optimal policy**  $\pi^*$  viene dada por realizar la acción que tiene un valor mayor de  $Q^*$  para el estado en el que estamos.

Por otro lado, la función  $Q^*$  satisface la **ecuación de Bellman**:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Figura 119. Ecuación de Bellman.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

De manera intuitiva, esta ecuación nos dice que si los valores de la función óptima  $Q^*$  son conocidos en  $Q^*(s', a')$ , entonces la estrategia óptima consiste en encontrar la acción que maximiza el valor de  $Q^*$  para cualquier de las acciones  $a'$  que podemos ejecutar después de  $(s, a)$ . Esto es, partiendo de todas las acciones que podemos hacer a partir de ejecutar  $a$  en el estado  $s$ , si conocemos el valor óptimo para cada posible acción  $a'$  en el estado  $s'$  en el que hemos acabado, hemos de tomar la que maximice el valor. La esperanza aparece de nuevo porque en el marco de los MDP el estado destino tiene cierto componente aleatorio a partir de la distribución  $P_{sa}$ .

La ecuación de Bellman nos da un primer método para obtener  $Q^*$  y que es conocido como **value iteration**, consiste en utilizar la ecuación de Bellman de forma iterativa:

$$Q_{i+1}(s, a) = \mathbb{E} \left[ r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

Figura 120. Value iteration.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

De manera que  $Q$  converge a  $Q^*$  cuando  $i$  tiende a infinito.

Este método funciona correctamente y ha sido muy utilizado en la época anterior a la llegada del *deep learning*. Si bien no profundizaremos en el algoritmo en clase, comentar que este tiene el problema de que necesita calcular  $Q(s, a)$  para cada pareja (estado, acción). Esto lo hace solo tratable computacionalmente para problemas con un número de estados y acciones muy reducidos. Por ejemplo, en un problema donde un agente intente aprender a jugar a videojuegos, un estado viene dado por todos los píxeles de la pantalla, lo cual hace computacionalmente imposible mantener una tabla con valores para cada estado.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 8.4. Deep Q-Learning



Accede al vídeo «Deep Q-Learning» a través del aula virtual

Hasta ahora hemos visto el marco tradicional de *reinforcement learning*. En este apartado veremos uno de los grandes avances en esta área obtenido a partir de los avances en *deep learning*. En particular, veremos cómo funciona el *Deep Q-Learning* y cómo fue utilizado por la empresa DeepMind para hacer que unos agentes aprendieran a jugar a juegos clásicos de Ataria directamente a partir de las imágenes del juego.

Anteriormente, hemos visto que el algoritmo de *value iteration* para *Q-learning* es computacionalmente intratable para problemas con un gran número de estados o acciones. La idea para superar este problema es utilizar una aproximación en forma de función para estimar la *Q-function*. De este modo, si obtenemos una función aproximada, no necesitamos tratar el problema de manera discreta por cada combinación de estado y acción.

$$Q(s, a; \theta) \approx Q^*(s, a)$$

Figura 121. Función aproximada para estimar la *Q-funciton*.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

La idea clave del Deep Q-learning (idea que seguro no nos sorprende a estas alturas) es utilizar una **red neuronal profunda como función de aproximación**.

De nuevo, partiremos de que la *optimal Q-function* que buscamos satisface la ecuación de Bellman:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Figura 122. Optimal Q-funciton.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

Nuestra red neuronal calculará  $Q(s, a; \theta)$  en donde:

- ▶ La red tendrá de *input* el estado  $s$  y en la última capa tendrá una salida por cada posible acción  $a$ .
- ▶  $\theta$  representa en este caso los parámetros o *weights* de la red.

Para aprender esta red neuronal, definiremos de nuevo una función de pérdida o *loss function*:

$$L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$$

Figura 123. Función de pérdida.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

Esta función nos dice lo bien que aproxima la red  $Q(s, a; \theta)$  al valor real  $y_i$ . Sin embargo,  $y_i$  indica el valor óptimo que buscamos, y este valor es desconocido!

Así que utilizaremos la ecuación de Bellman de nuevo para obtener este valor ( $y_i$ ) de manera iterativa a partir de la red neuronal actual:

$$y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$$

Figura 124. Función de pérdida.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

Nótese que  $y_i$  es la parte de la derecha de la ecuación de Bellman, donde hemos sustituido  $Q^*$  por la aproximación  $Q(s, a; \theta_{i-1})$ , esto es, la aproximación que teníamos en un valor de iteración anterior. Así, de manera iterativa intentaremos hacer que el *Q-value* se acerque al valor que debería de tener ( $y_i$ ) según la ecuación de Bellman si  $Q$  fuese óptimo.

### Caso de estudio con *Atari Breakout*

Veamos un caso particular del algoritmo de *deep Q-learning* aplicado al juego de *Atari Breakout*. Este fue uno de los juegos de Atari utilizados para entrenar agentes en el *paper* original de DeepMind. En gran parte de los juegos, los agentes entrenados con aprendizaje por refuerzo consiguieron alcanzar un nivel mucho más elevado que el de los humanos.

En *Atari Breakout* el objetivo es romper todos los ladrillos de la pantalla usando una bolita que controlamos con una paleta inferior de manera que no caiga al vacío. El objetivo es acabar el juego con la mayor puntuación posible. El estado viene dado por todos los píxeles de la pantalla y las acciones son mover la paleta a la izquierda y a la derecha. Las recompensas son puntos según los ladrillos rotos, y una recompensa negativa a lo largo del tiempo para forzar al agente a acabar cuanto antes.



Figura 125. Atari Breakout.

Fuente: [https://www.researchgate.net/figure/Screenshot-of-the-Atari-Breakout-game\\_fig16\\_324780469](https://www.researchgate.net/figure/Screenshot-of-the-Atari-Breakout-game_fig16_324780469)

En la siguiente figura, vemos la arquitectura de la *Q-network* que define el algoritmo de *Q-learning*.

- ▶ El *input state* se corresponde con 4 imágenes 84x84 con 4 frames del juego.
- ▶ Arriba de este hay varias capas convolucionales que se encargan de encontrar una representación de la pantalla en forma de *features* de alto nivel, de modo que el algoritmo sea capaz de aprender conceptos como la bola, la paleta y los ladrillos.
- ▶ Más arriba, tenemos dos *fully connected layers*, la última de ellas con una salida por cada acción posible, que depende del juego en particular. La acción también podría pasarse como *input* a la red, pero si el número de acciones no es muy grande, es más eficiente obtener los *Q-values* de cada acción en el mismo *forward pass*.

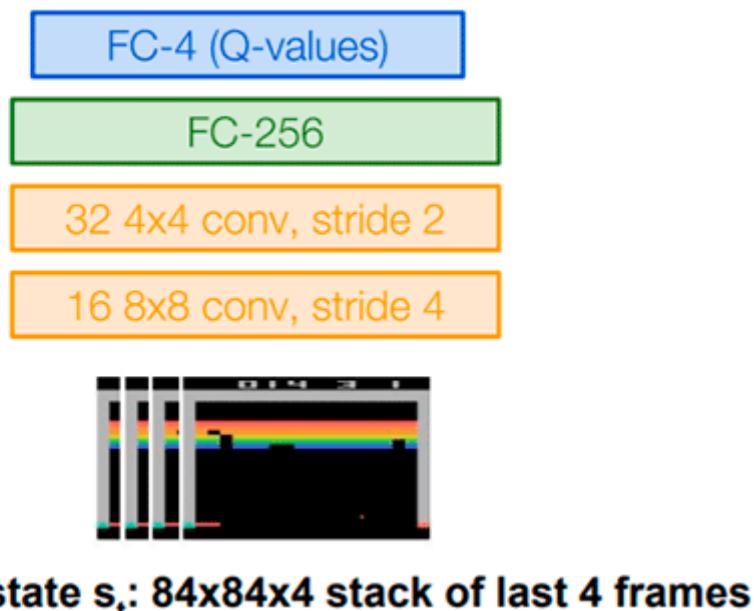


Figura 126. *Q-network* para *Breakout*.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture14.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf)

Para el entrenamiento de la red, se utilizó una serie de trucos explicados en el *paper*. Uno de ellos es ***Experience Replay***. En vez de simplemente jugar y alimentar a la red neuronal con *inputs* consecutivos, se crea una memoria de experiencias conocidas (acciones, transiciones y recompensas) y se crean mini *batches* de manera aleatoria

a partir de ella. Esto evita tener *samples* del juego correlacionadas y hace el entrenamiento mucho más efectivo.

Por otro lado, hemos visto que la propia red  $Q(s, a; \theta)$  se utiliza en el objetivo  $y_i$ , y por tanto cambia de manera continua. Para hacer el proceso de entrenamiento más estable, la red en el objetivo  $y_i$  solo se actualiza con los valores actuales de la *Q-network* cada 1000 *steps*.

Veamos el algoritmo completo de *Deep Q-learning* con *Experience Replay*:

---

**Algorithm 1** Deep Q-learning with Experience Replay

---

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
    end for
end for

```

---

Figura 127. Algoritmo de *Deep Q-learning*.

Fuente: <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>

1. El primer paso es inicializar la *replay memory* y la *Q-network* con pesos aleatorios.
2. A partir de ahí, se juegan  $M$  episodios (partidas completas). Por cada episodio, se obtiene un estado inicial y una secuencia preprocesada, lo que quiere decir que los 4 frames de las imágenes son procesados mediante varias técnicas, como convertir a blanco y negro.
3. Después, a lo largo de  $T$  instantes de tiempo en cada partida, se selecciona una acción al azar o se escoge la mejor acción posible a partir de la *Q-function* actual. La elección de una acción al azar se corresponde a que en ocasiones queremos explorar nuevos caminos para ver si encontramos alguna solución mejor a lo que

tenemos hasta ahora, y es un problema clave en el mundo del aprendizaje por refuerzo (*exploration-exploitation tradeoff*).

4. Una vez elegida la acción a realizar, se ejecuta esta acción en el emulador del juego, lo cual nos da el siguiente estado y la recompensa, que se almacenan en la *replay memory*. En este punto es cuando realmente entrenamos la Q-network, obteniendo una *batch* aleatoria de elementos a partir de la memoria y ejecutando *gradient descent*.



Accede a los ejercicios de autoevaluación a través del aula virtual

---



Accede al vídeo «Resumen» a través del aula virtual

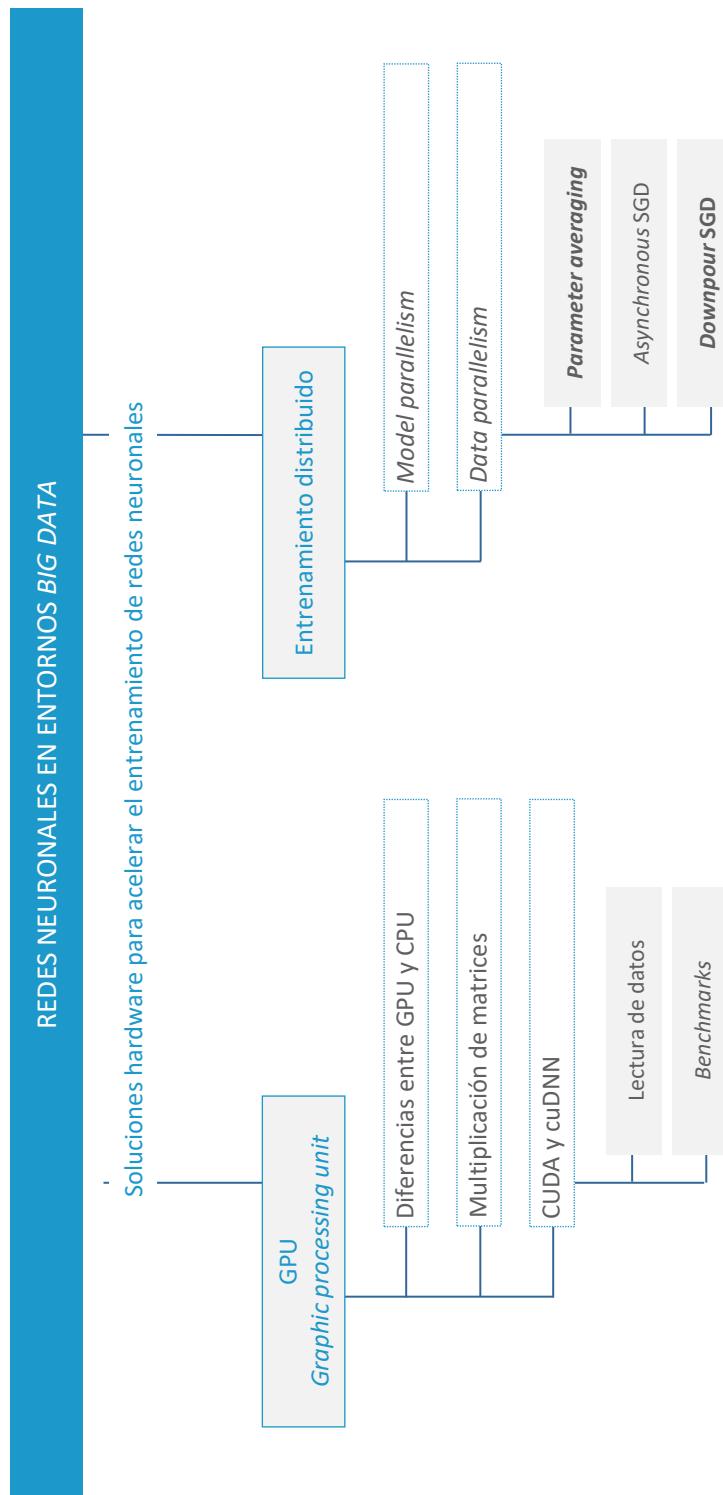
---

# Tema 9. Redes neuronales en entornos *Big Data*

## Índice

Esquema	3
Ideas clave	4
9.1. Introducción y objetivos	4
9.2. GPU para entrenamiento de redes neuronales profundas	4
9.3. Entrenamiento distribuido	12

# Esquema



## 9.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

En este tema veremos cómo acelerar el entrenamiento de redes neuronales con GPU y en sistemas distribuidos. Esto resulta fundamental para el entrenamiento de redes neuronales gigantes en entornos *Big Data*, donde la cantidad de datos disponibles es inmensa.

Es importante comprender en este tema por qué las GPU son capaces de acelerar el entrenamiento de los algoritmos de *deep learning*, así como las diferencias con las CPU que resultan claves para esto. Asimismo, es importante asimilar por qué factores como la velocidad de red o el ancho de banda de memoria influyen en la efectividad del uso de GPU y sistemas distribuidos. Otro punto esencial en este tema son los conceptos de *model parallelism* y *data parallelism*, así como entender qué ventajas y desventajas tienen algunas implementaciones particulares.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 9.2. GPU para entrenamiento de redes neuronales profundas



Accede al vídeo «GPU para entrenamiento de redes neuronales profundas» a través del aula virtual

Como comentamos al principio del curso, la gran disponibilidad de datos y la mejora de la capacidad de cómputo para entrenar redes neuronales han sido dos de los factores que han llevado al éxito del *deep learning*. En el aspecto de la capacidad de computación, la utilización de GPU para optimizar el proceso de entrenamiento ha jugado un papel clave. Esta **mejora en el proceso de entrenamiento** ha permitido que la experimentación y la investigación con redes neuronales haya sido más rápida y efectiva, haciendo posible, además, la resolución de problemas casi intratables hasta entonces por la escala de tiempo necesaria. Por ejemplo, utilizando GPU, el tiempo que los investigadores necesitaban para entrenar la primera AlexNet era de alrededor de una semana.

Una tendencia clara en el mundo del *deep learning* a lo largo de los años es que los modelos se hacen más grandes y profundos, necesitando más operaciones para ser entrenados y, del mismo modo, aprovechando mejor los datasets mayores que van apareciendo. Por ello, alcanzar un entrenamiento rápido y eficiente de redes neuronales es un tema de gran importancia. En esta ocasión nos centraremos en los avances de hardware y de sistemas distribuidos para conseguir entrenar de manera eficiente en situaciones donde la cantidad de datos es muy grande, pudiéndose hablar de *Big Data*.

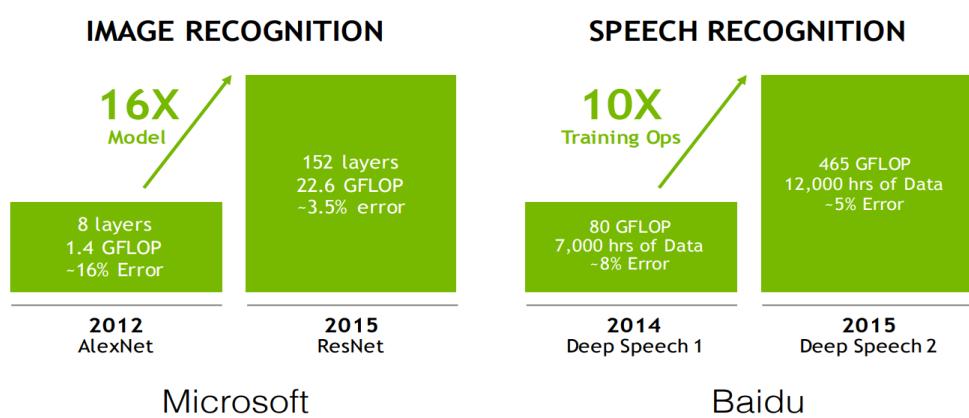


Figura 128. Ejemplo del aumento de modelos y operaciones para *Image* y *Speech Recognition*.

Fuente: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture15.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf)

## GPU: *graphics processing unit*

Una GPU (*graphics processing unit*) es un coprocesador hardware dedicado al procesamiento de gráficos y operaciones de coma flotante, con el objetivo de aligerar la carga de trabajo del procesador. Tradicionalmente, las GPU se han utilizado para videojuegos o aplicaciones gráficas con efectos avanzados, por ejemplo 3D. Visto así, no es más que una tarjeta gráfica de toda la vida, si bien en el mundo del *deep learning* es preponderante la denominación GPU.



Figura 129. GTX 1080 Ti.

Fuente: <https://www.evga.com/articles/01092/evga-geforce-gtx-1080-ti/>

La labor de las GPU es, por tanto, complementar a la CPU en cierto tipo de operaciones para las cuales han sido diseñadas y optimizadas. Veamos **en qué se diferencian una CPU y una GPU**, tomando como ejemplo un procesador Intel i7-7700k y una NVIDIA GTX 1080 Ti.

Una primera diferencia es que las CPU suelen tener entre 2 y 10 *cores* o núcleos, esto es, son capaces de ejecutar en paralelo entre 2 y 10 procesos o hilos (en el ejemplo, 8 hilos con *hyperthreading*). La velocidad de reloj para estos puede llegar a ser de más de 4GHz.

Por otro lado, una tarjeta gráfica tiene varios miles de cores que corren, sin embargo, a una velocidad de reloj inferior.

	Cores	Clock Speed	Memory	Price	Speed
<b>CPU</b> (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	\$339	~540 GFLOPs FP32
<b>GPU</b> (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	\$699	~11.4 TFLOPs FP32

Figura 130. CPU vs. GPU, hardware de 2017-2018.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture08.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf)

- ▶ Como vemos en la diferencia de *cores*, una GPU puede ejecutar un número de tareas en paralelo mucho mayor que una CPU. Esta diferencia viene dada por el **tipo de trabajo** para el que cada cual ha sido diseñada:
  - Las CPU resuelven tareas de propósito general, mientras que las GPU están optimizadas para tareas muy particulares.
  - Las GPU tienen muchos *cores*, pero cada uno de estos es más lento y limitado a unas pocas operaciones, mientras que las CPU tienen pocos núcleos pero muy rápidos y capaces de realizar un gran número de tareas.

De este modo, las GPU destacan en su capacidad de parallelizar de manera masiva operaciones sencillas, mientras que las CPU están más orientadas a tareas secuenciales generales.

- ▶ Otra gran diferencia es el **uso de memoria**. Mientras que las CPU utilizan la memoria RAM del sistema, las GPU tienen su propia memoria RAM integrada.
  - La cantidad de memoria disponible en una GPU ha ido en aumento durante los últimos años, lo cual ha permitido entrenar modelos más grandes. El modelo (los parámetros de la red) tiene que estar en la memoria de la GPU, lo cual supone en muchas ocasiones una limitación. Esto lleva a sistemas complejos

donde los modelos se dividen y entranen entre varias GPU, como vimos en AlexNet.

- Asimismo, las GPU están optimizadas para obtener grandes cantidades de memoria de manera rápida (tienen un mayor ancho de banda, *memory bandwidth*). Mientras que las CPU son buenas en utilizar pequeñas cantidades de memoria de manera muy rápida, las GPU destacan en leer grandes cantidades de memoria de manera eficiente, si bien con mayor latencia.

Esto hace de nuevo que las tarjetas gráficas sean más efectivas para grandes operaciones numéricas como son las redes neuronales (recordemos que podemos tener millones de parámetros en una red).

### Multiplicación de matrices en GPU

Veamos ahora, a modo de ejemplo, una **operación altamente paralelizable** que puede ejecutarse de manera muy eficiente en una tarjeta gráfica: la multiplicación de matrices.

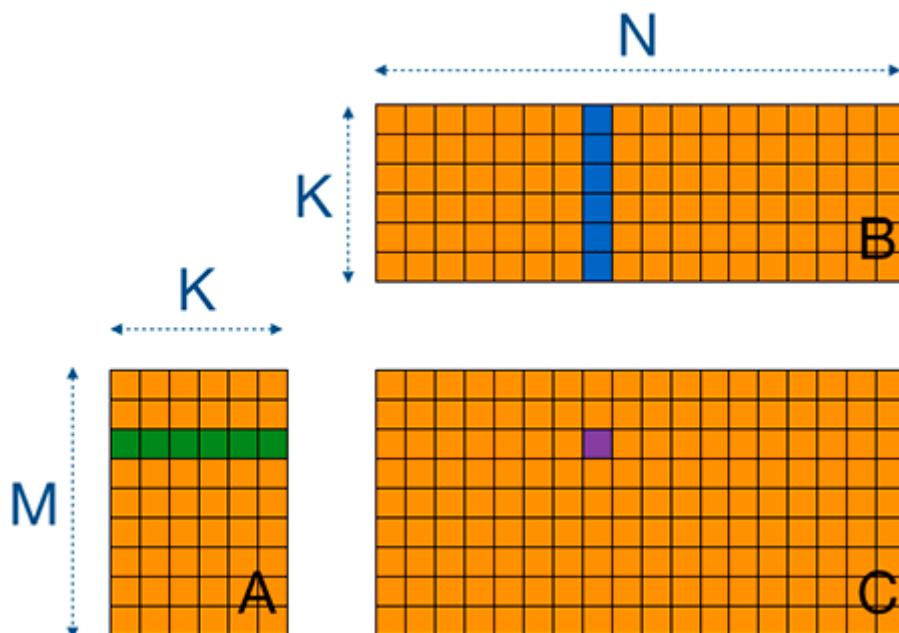


Figura 131. Multiplicación de matrices.

Fuente: <https://cnugteren.github.io/tutorial/pages/page2.html>

Como sabemos, cada elemento en la matriz resultante es el producto escalar de una fila y una columna de las matrices a multiplicar. Esto hace que podamos calcular cada elemento de la matriz de salida de manera paralela, ya que no se necesita utilizar ningún valor intermedio.

Una GPU puede leer de manera rápida matrices gigantescas de una manera eficiente, pues tienen un gran ancho de banda de memoria, y utilizar sus miles de núcleos (otra gran diferencia respecto a las CPU) para **obtener de manera paralela todos los valores de salida**. Las operaciones a realizar son muy sencillas, ya que se trata simplemente de multiplicaciones y sumas de números reales, lo cual entra en el catálogo de operaciones relativamente simples que una GPU puede realizar.

A diferencia de la GPU, una CPU iría elemento a elemento de manera secuencial (o de 8 en 8, si dedicamos los 8 *cores* a 8 hilos distintos). El número de *cores* de una CPU se aleja bastante, como hemos visto, de los miles de cores disponibles en una GPU, por lo que es fácil ver de dónde viene la mejora de velocidad. Sin embargo, este ejemplo está en cierta modo muy simplificado, ya que las CPU actuales suelen tener operaciones vectorizadas y librerías algebraicas de alto rendimiento capaces de realizar multiplicaciones de matrices y otras operaciones numéricas de manera muy eficiente y utilizando varios procesadores.

Este ejemplo de la efectividad de las GPU calculando multiplicaciones de matrices es clave para entender por qué son tan útiles para el *deep learning*. Las redes neuronales, al fin y al cabo, son una serie de multiplicaciones de matrices. Todas las operaciones que hemos visto, capa a capa con los parámetros  $w$  de una red, pueden efectuarse en forma de producto de matrices. En muchos casos, productos de matrices enormes, ya que hay redes donde las capas pueden tener cientos o miles de elementos.

Del mismo modo, operaciones típicas como las convoluciones pueden ser también paralelizadas de manera sencilla. Todo esto permite que una GPU acelere el proceso de entrenamiento de una red neuronal en gran medida.

## CUDA y cuDNN

De los dos fabricantes clásicos de tarjetas, NVIDIA y AMD, el primero se ha destacado en los últimos años por apostar fuerte en la utilización de sus GPU para *deep learning*, desarrollando incluso hardware optimizado para este fin. Una de las claves de su éxito ha sido CUDA, una plataforma de desarrollo para computación paralela con GPU. CUDA permite sacar el máximo provecho de las capacidades paralelas de las tarjetas gráficas, para lo que utiliza su propio lenguaje de programación, muy parecido a C.

Si bien desarrollar código CUDA no es sencillo, NVIDIA ofrece una serie de librerías y API que permiten sacar provecho de las capacidades de computación paralela de las GPU; una de estas es cuDNN, especialmente orientada a *deep learning*.

cuDNN implementa primitivas altamente optimizadas para el entrenamiento de redes neuronales profundas, tales como *forward* y *backward passes* de operaciones que hemos visto en este curso, como *dense layers*, *convolutions*, *pooling*, *batch normalization*, RNN, etc.

Librerías como cuDNN permiten a los desarrolladores centrarse en definir y entrenar modelos, facilitándoles abstraerse completamente de la complejidad de optimizar código para GPU. Casi todos los *frameworks* de *deep learning* que hemos visto en este curso, tales como TensorFlow y PyTorch, son capaces de acelerar sus operaciones con cuDNN, si esta librería está instalada y se dispone de una tarjeta gráfica NVIDIA. Aquí vemos de nuevo la ventaja que supone definir los *frameworks* en forma de un grafo de computación:

**Las distintas operaciones del grafo pueden aprovechar una implementación más eficiente en GPU a partir de librerías de más bajo nivel como cuDNN.**

CUDA y cuDNN solo funcionan en GPU de NVIDIA. Para tarjetas de ATI o para elementos de hardware más generales, existe una alternativa abierta llamada OpenCL.

## Lectura de datos

La mejora de rendimiento que conlleva el uso de GPU puede provocar en ocasiones que el nuevo cuello de botella, en vez de ser las complejas operaciones matemáticas del entrenamiento de redes neuronales, sea la lectura y el procesamiento de los datos que utiliza la red para entrenar. El modelo está guardado en la RAM de la GPU, pero los datos suelen venir de ficheros en el disco duro.

Por ello, **para maximizar la cantidad de datos** que se puede alimentar a la red, conviene tener en cuenta ciertos factores:

- ▶ Si es posible, leer todos los datos a usar en la RAM de la máquina, de manera que la GPU tenga un acceso rápido a ellos.
- ▶ Utilizar un disco duro de estado sólido (SSD) en vez de un disco duro clásico (HDD). Los SSD son mucho más rápidos para lectura/escritura.
- ▶ Utilizar varios hilos de la CPU para leer y procesar datos. Esto es especialmente importante si necesitamos procesar el *training data* antes de pasarlo a nuestro algoritmo de aprendizaje. Por ejemplo, *tokenizar* el texto (separarlo en palabras), hacer transformaciones sobre imágenes, etc. Si utilizamos varios hilos de la CPU, podemos tener los datos preparados en un *buffer*, todo esto ocurriendo en paralelo al proceso de entrenamiento. El objetivo es conseguir que la GPU no se quede nunca esperando a que los datos estén listos.

## Benchmarks

Es muy frecuente encontrar *benchmarks* donde se compara la velocidad de entrenamiento con distintos *frameworks* de *deep learning*, así como el uso de distintas librerías de optimización y tarjetas gráficas. Con estos *benchmarks*, es fácil ver cómo las GPU optimizan el proceso de entrenamiento de manera impresionante.

Como ejemplo, en la siguiente gráfica podemos ver los tiempos de entrenamiento para varias arquitecturas profundas utilizando solo CPU, GPU sin cuDNN y finalmente GPU con cuDNN.

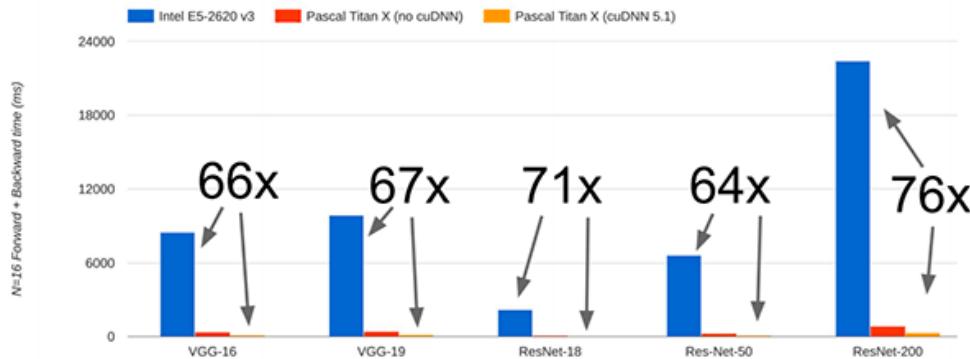


Figura 132. Tiempos de entrenamiento utilizando CPU, GPU sin cuDNN y con cuDNN.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture08.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf)



Accede a los ejercicios de autoevaluación a través del aula virtual

## 9.3. Entrenamiento distribuido



Accede al vídeo «Entrenamiento distribuido» a través del aula virtual

En el apartado anterior, hemos visto cómo escalar el entrenamiento de redes neuronales con una gran cantidad datos en una sola máquina mediante la utilización de GPU. Sin embargo, ¿qué pasa si con esto no es suficiente? Una vez que ya no podemos hacer nuestro entrenamiento más rápido a base de tener una CPU y una o varias GPU más caras y potentes, no queda más remedio que empezar a buscar soluciones donde el entrenamiento se distribuya a través de varias máquinas.

Un sistema distribuido nos permite utilizar los recursos de muchas máquinas a la vez. Incluso en ocasiones puede ser más rentable utilizar varios servidores baratos en vez

de solo uno con los últimos avances disponibles. Sin embargo, la utilización de sistemas distribuidos conlleva varias complejidades que hay que tener en cuenta:

- ▶ Los algoritmos de entrenamiento diseñados para un funcionamiento secuencial tienen que ser adaptados a un entorno donde varias máquinas hacen cálculos a la vez. En nuestro caso, si distribuimos el cálculo de *stochastic gradient descent* a varias máquinas, este tiene que ser adaptado de alguna manera.
- ▶ Las máquinas del sistema distribuido han de tener cierta comunicación y coordinación entre ellas. Esto añade complejidad al sistema y nos obliga a tener en cuenta los tiempos de transferencia de datos por red, siempre más altos que el movimiento de datos en memoria RAM dentro de una sola máquina.
- ▶ El hecho de tener varias máquinas implica que la probabilidad de que una de ellas falle es más alta. Esto implica a su vez nuevas dificultades, ya que nos gustaría disponer de sistemas donde el entrenamiento no se detenga completamente si una máquina se cae.

### *Model parallelism y data parallelism*

Para empezar, vamos a definir dos enfoques posibles para paralelizar o distribuir el entrenamiento de redes neuronales.

#### *Model parallelism*

En este enfoque, el modelo se distribuye en partes y cada máquina del sistema distribuido se hace responsable de los cálculos de una parte de la red, enviando los resultados correspondientes a otras máquinas. Por ejemplo, cada máquina podría realizar los cálculos de cada capa de una red.

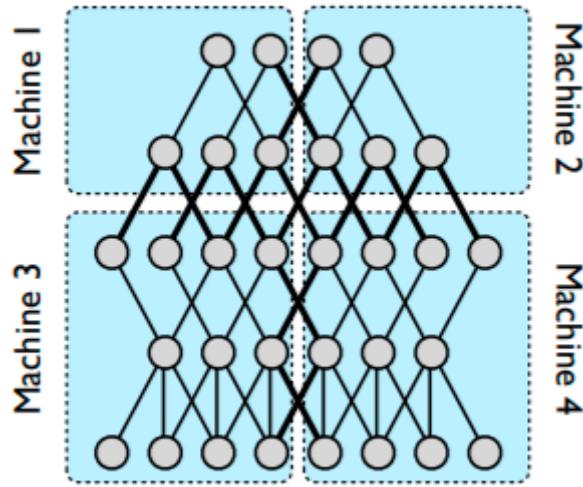


Figura 133. *Model parallelism*.

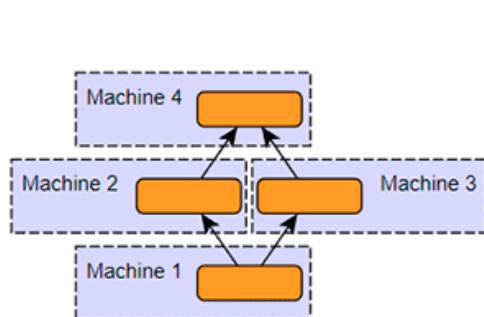
Fuente: <http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf>

En la imagen de arriba podemos ver cómo una red se parte en cuatro máquinas distintas. En general, cualquier grafo de computación puede partirse de manera similar. Las aristas en negrita denotan una transferencia de datos entre máquinas.

### ***Data parallelism***

Por otro lado, en *data parallelism* cada máquina tiene una copia completa del modelo y un subconjunto distinto de los datos de entrenamiento. Cada una va entrenando el modelo con sus datos y los distintos modelos resultantes son combinados cada cierto tiempo. En las siguientes secciones veremos ejemplos de cómo se hace esto.

Model Parallelism



Data Parallelism

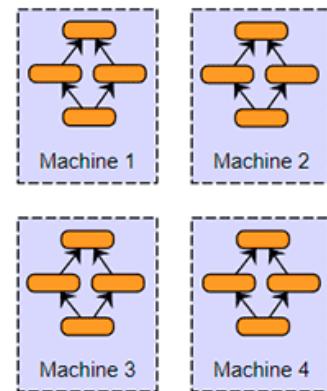


Figura 134. *Model parallelism vs. data parallelism*.

Fuente: <https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-of-neural-networks/>

Estos dos enfoques no se excluyen mutuamente. Por ejemplo, podríamos tener un clúster de máquinas en modo *data parallelism* donde cada máquina distribuye el modelo en varias GPU.

Model and Data Parallelism

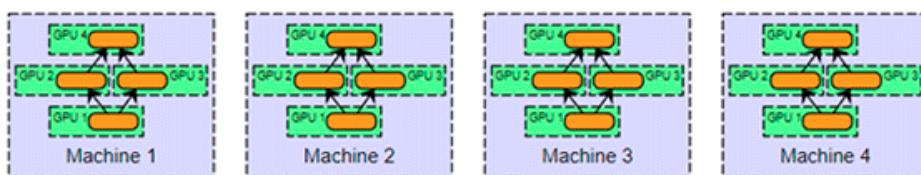


Figura 135. Híbrido de *Model parallelism* y *data parallelism*.

Fuente: <https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-of-neural-networks/>

Si bien *model parallelism* permite distribuir modelos que no caben en una sola máquina, el **esquema preferido** para el entrenamiento distribuido es *data parallelism*. Esto se debe a que es más sencillo de implementar, con una tolerancia a errores más fácil de tratar y con (normalmente) una utilización más efectiva de los recursos.

## *Parameter averaging*

Es una de las formas más sencillas de aplicar *data parallelism*. En *parameter averaging* tenemos un servidor de parámetros (*parameter server*) encargado de mantener la versión actual del modelo a partir del trabajo de las distintas máquinas entrenando el modelo o *workers*. El proceso es el siguiente:

8. Los parámetros se inicializan de manera aleatoria siguiendo una de las estrategias vistas en clase.
9. El *parameter server* distribuye una copia de los parámetros actuales a cada *worker*.
10. Cada *worker* realiza el entrenamiento de una o varias *training batches* con su subconjunto de datos.
11. Cada *worker* envía los nuevos parámetros del modelo que ha obtenido al entrenar al *parameter server*.
12. El *parameter server* espera a recibir todos los parámetros de todos los *workers*. Una vez recibidos, se establecen los nuevos parámetros actuales del modelo como la media (*average*) de todos los parámetros recibidos.
13. Se vuelve al punto 2 y se repite el proceso durante el número de épocas (*epochs*) deseado.

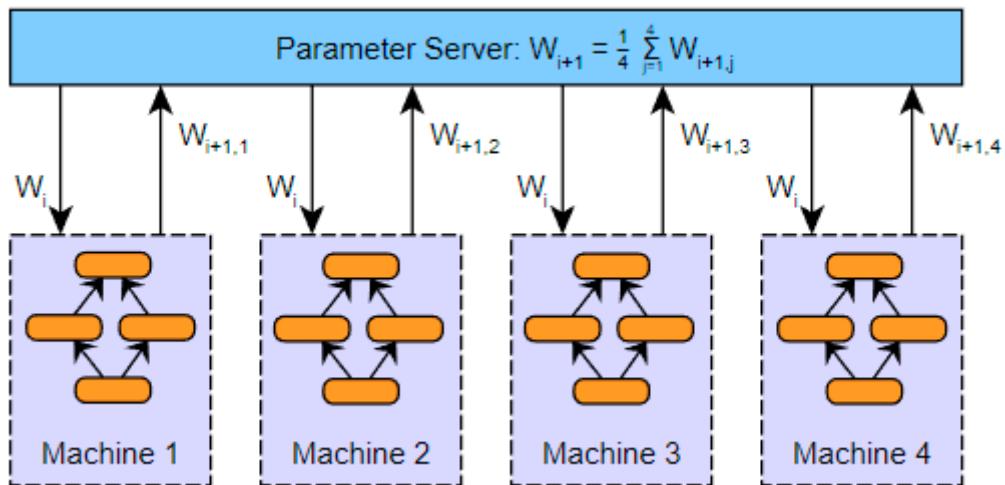


Figura 136. *Parameter averaging*.

Fuente: <https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-of-neural-networks/>

Uno de los aspectos clave a definir en este método es cada cuántas iteraciones se hace la media de los parámetros. Lo mejor sería hacerlo por cada iteración (*batch*), pero esto puede acabar haciendo que el coste de transferir los datos por la red al *parameter server* domine totalmente al coste de computación. Por otro lado, no se puede elegir un número muy grande de iteraciones, ya que cada máquina podría estar convergiendo a una solución distinta, con el peligro de que la media no dé lugar a un buen conjunto de parámetros.

Asimismo, *parameter averaging* tiene que esperar a que el trabajador (*worker*) más lento termine para hacer el *update* del modelo, lo que implica que acabamos perdiendo recursos mientras los otros trabajadores esperan. Del mismo modo, si uno de ellos falla, el proceso se detiene por completo, salvo que definamos alguna medida de protección como una media entre los trabajadores restantes.

### *Asynchronous SGD y Downpour SGD*

#### **Asynchronous SGD**

Esta es una versión para ejecución en paralelo de stochastic gradient descent. Como sabemos, SGD actualiza los parámetros de la red utilizando los gradientes de la función de coste a partir de *batches* de elementos de *training data* al azar.

Una forma de paralelizar SGD sería similar a la vista en *parameter averaging*, donde los distintos *workers* se sincronizan y van enviando sus *updates* (en este caso, los gradientes calculados en vez de los parámetros obtenidos). Esto, como hemos visto, produce ciertas ineficiencias en tanto que los *workers* tienen que esperar y sincronizarse entre ellos, en vez de actuar de manera independiente.

Una alternativa a esto es convertir el entrenamiento por SGD en paralelo en una especie de salvaje oeste donde cada *worker* calcula sus gradientes, los envía al servidor de parámetros y recoge de allí los nuevos parámetros calculados, probablemente conteniendo a su vez los *updates* de otros *workers*. La clave aquí es que el trabajador no tiene que esperar a que los otros terminen el proceso, sino que envía sus gradientes una vez ha terminado el cálculo y prosigue una vez recibe los parámetros actualizados.

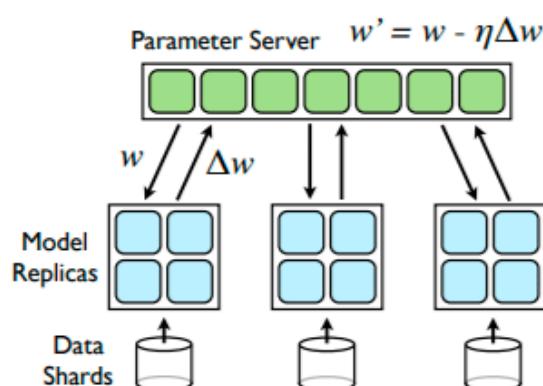


Figura 137. Downpour SGD.

Fuente: <http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf>

### Downpour SGD

En *Downpour SGD*, otro sistema de distribución basado en *data parallelism*, el servidor de parámetros va recibiendo los gradientes de cada *worker* de manera asíncrona. Cuando los recibe, ejecuta la ecuación de SGD y obtiene los nuevos parámetros, que a su vez envía al *worker* que acaba de enviar los gradientes, ya que este necesita conocer cuál es el nuevo estado del modelo para seguir calculando. Como vemos, este proceso evita tener que esperar a los trabajadores lentos e, igualmente, no implica problemas si un trabajador falla.

Puede parecer extraño que un método así funcione. Al fin y al cabo, los *workers* están trabajando con parámetros del modelo que cambian continuamente y que, además, se quedan desactualizados cuando otros *workers* entregan sus *updates*. Este fenómeno, conocido como ***stale gradient problem***, puede dar lugar a problemas durante el entrenamiento. En la práctica, se aplican ciertas soluciones y el entrenamiento mediante *Asynchronous SGD* suele ser muy efectivo y converger de manera adecuada.

*Downpour SGD* fue implementado por Google en DistBelief, el precursor de TensorFlow, y en la actualidad se utiliza también en sistemas de producción. Algo que se hace de manera común cuando el número de *workers* es elevado es utilizar varios *parameter servers* en vez de uno.

Cada *parameter server* contiene parte de los parámetros del modelo para evitar que una sola máquina reciba los *updates* de todos los trabajadores. Como sabemos, los modelos de *deep learning* pueden llegar a tener millones o billones de parámetros, por lo que se puede producir un cuello de botella en la interfaz de red de la máquina encargada de recibir parámetros de todos los *workers*.

La distribución del *parameter server* en varias máquinas también puede crear problemas, ya que la caída de uno solo de ellos implica parar el entrenamiento hasta que el servidor vuelva. De manera similar, es importante distribuir los parámetros de la red de manera adecuada entre los *parameter servers*. Por ejemplo, si estos servidores contienen *word embeddings* como parámetros y las palabras más

comunes se guardan en el mismo servidor, este recibirá *updates* de manera mucho más frecuente, lo que puede colapsar su interfaz de red o crear un cuello de botella en la CPU al aplicar SGD para esos parámetros.



Accede a los ejercicios de autoevaluación a través del aula virtual

---



Accede al vídeo «Resumen» a través del aula virtual

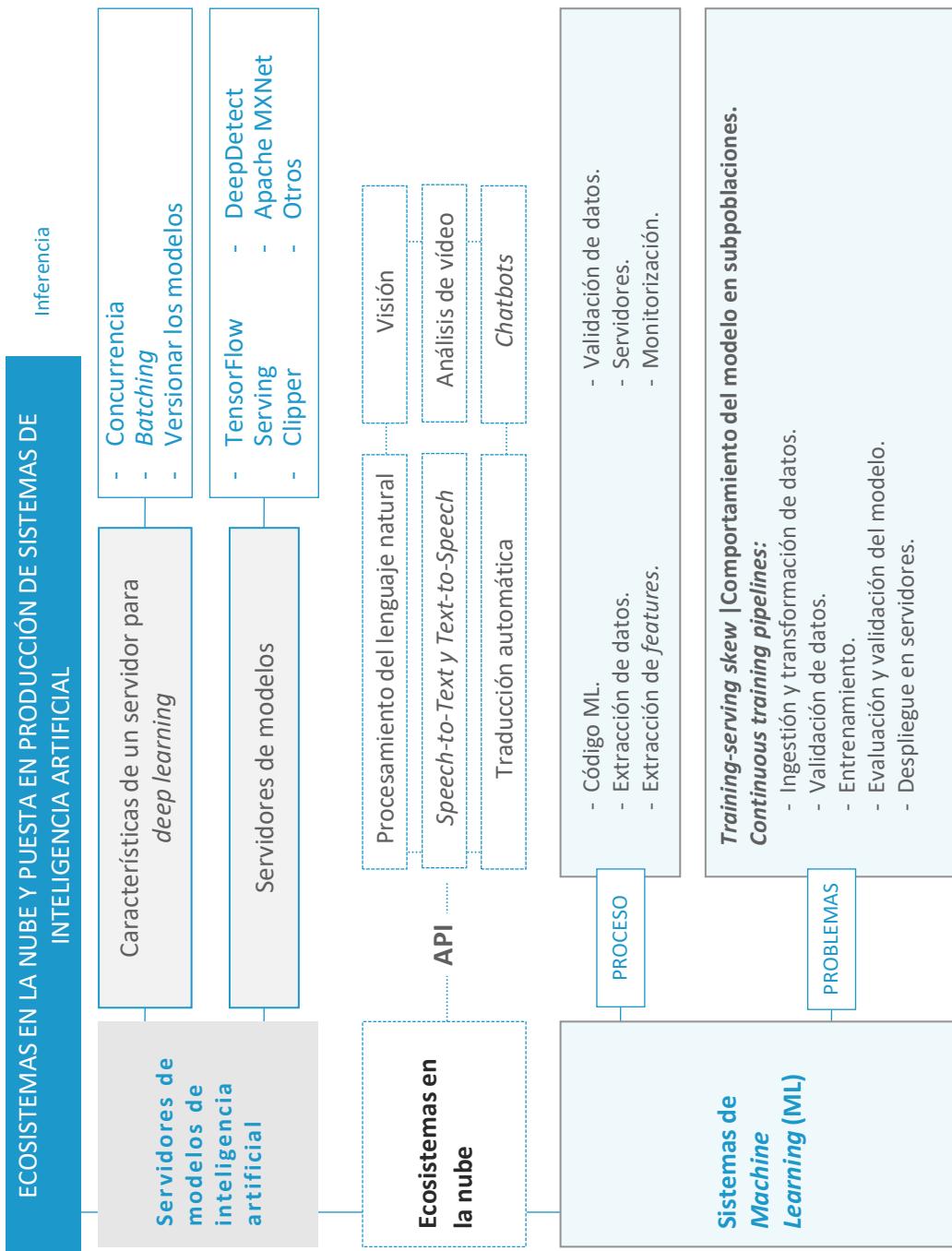
---

# Tema 10. Ecosistemas en la nube y puesta en producción de sistemas de inteligencia artificial

## Índice

Esquema	3
Ideas clave	4
10.1. Introducción y objetivos	4
10.2. Servidores de modelos de inteligencia artificial	4
10.3. Ecosistemas en la nube	9
10.4. Aspectos prácticos de la puesta en producción de sistemas de <i>machine learning</i>	11

# Esquema



## 10.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

En este tema veremos una serie de aspectos importantes a la hora de poner en uso un sistema de *machine learning*. Esto es más complicado de lo que podría parecer y requiere la utilización de una infraestructura para servir modelos.

Es importante comprender por qué es necesario tener servidores para poner en producción un sistema de *machine learning* y qué características suelen tener estos. Asimismo, también es esencial entender qué es un ecosistema en la nube y el papel de los proveedores de *cloud computing*. Finalmente, hay que conocer los aspectos de la puesta en producción de un sistema de aprendizaje automático y cuáles son algunos de los problemas más comunes a los que hay que estar atento.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 10.2. Servidores de modelos de inteligencia artificial



Accede al vídeo «Servidores de modelos de inteligencia artificial» a través del aula virtual

Hasta ahora nos hemos centrado en el entrenamiento de redes neuronales profundas. Sin embargo, otro aspecto fundamental es utilizar estos modelos para

**inferencia**, esto es, utilizar los modelos ya entrenados para extraer conocimiento a partir de nuevos datos. El proceso de utilizar datos no vistos durante el entrenamiento en un modelo es comúnmente llamado inferencia o *inference* (a diferencia del entrenamiento/aprendizaje), *serving time* (a diferencia de *training time*) y, en ocasiones, *test time* o *prediction time*.

Durante el proceso de inferencia, alimentamos la red neuronal con datos y recogemos la salida de la red para extraer la información deseada. Por ejemplo, en una red convolucional que clasifique imágenes, podemos obtener la clase resultante a partir de las probabilidades de la última capa.

Utilizar una red para inferencia es una tarea computacionalmente menos costosa que el aprendizaje. Al fin y al cabo, no tenemos que calcular *backpropagation* ni pasar varias veces por los mismos datos hasta converger. Basta con utilizar el *data point* a partir del cual queremos inferir y realizar un *forward pass* para obtener la salida de la red. Frameworks como TensorFlow o librerías como Keras permiten obtener los valores de salida directamente a partir de un modelo entrenado con una sola llamada.

Sin embargo, esto no significa que el proceso de inferencia no tenga sus desafíos y complejidades. El modelo se entrena una vez, pero podría ser utilizado miles o millones de veces con cientos o miles de accesos cada segundo. Imaginemos una red neuronal en un servicio de música o vídeo, calculando recomendaciones para sus usuarios, o un detector de personas que sugiere a quién etiquetar en una foto en una red social. De este modo, la puesta en producción de un modelo de inteligencia artificial presenta varias complicaciones en el plano ingenieril.

## Características de un servidor para *deep learning*

### Concurrencia

Una propiedad que nos gustaría satisfacer al servir un modelo de manera escalable es la concurrencia, esto es, poder **atender múltiples *requests*** (llamadas a nuestro modelo) a la vez. Normalmente, y dependiendo de la complejidad y tamaño del modelo, los tiempos para responder una solicitud no deberían ser mayores de medio segundo e, idealmente, deberían bajar de 100ms.

Los sistemas de *machine learning* son, en muchas ocasiones, complementarios a otras partes de un servicio (por ejemplo, las recomendaciones de artículos similares que podemos ver en una tienda *online* son una pequeña parte de toda la información que se nos presenta en pantalla) y, por tanto, no deberían de suponer un cuello de botella que retrase la respuesta al usuario. De este modo, si nuestro servidor solo permite atender una llamada o *request* al mismo tiempo, las llamadas se pueden ir acumulando y el último usuario en llegar tendría que esperar a que todos los anteriores sean respondidos. Esto daría lugar a un problema de **tiempos de respuesta**, en el que el *round-trip time* de una solicitud se alarga demasiado.

Un servidor concurrente permite tratar varias respuestas a la vez. Esto se consigue habitualmente mediante el uso de varios procesos o hilos, los cuales están a la espera de recibir y tratar *requests* según van llegando. Al llegar una, si un proceso está ocupado, otro proceso libre puede encargarse de la nueva solicitud, de modo que esta no tiene que esperar a la siguiente.

### ***Batching***

Otra característica deseable en un servidor para modelos de *deep learning* es el *batching*. Durante el curso, hemos visto el entrenamiento de redes neuronales utilizando *batches* de *training points* para el algoritmo SGD. De hecho, hacer un *batch* o lote de ejemplos no solo tiene un sentido matemático como aproximación al gradiente real, sino que suele **acelerar el entrenamiento de las redes neuronales** si se trata de manera adecuada.

Como sabemos, una red neuronal es calculada a través de multiplicaciones de matrices y operaciones optimizadas en forma vectorial. De este modo, si en vez de calcular las operaciones de la red, elemento a elemento, calculamos a la vez todos los elementos de la *batch* (pongamos el caso de 128), vamos a ganar velocidad al aprovechar operaciones optimizadas como la multiplicación de matrices. Esto es especialmente importante combinado con el uso de GPU que, como vimos en el tema anterior, aceleran en gran medida estas operaciones. Sin embargo, en inferencia las *requests* que recibimos, suelen ser elemento a elemento. ¿Cómo podemos aprovechar las ventajas computacionales del *batching*?

La solución consiste en ir guardando las *requests* que recibimos en un *buffer* y, cuando tengamos un número suficiente de ellas, crear una *batch* y pasarlal por nuestro modelo, obteniendo una salida con todos los valores deseados. Si, además, utilizamos una GPU en nuestro servidor, la aceleración será aún mayor.

Por supuesto, el *batching* solo es útil durante *serving time* y si recibimos un gran número de *requests* por segundo, ya que si solo llega una cada varios segundos, no tiene sentido dejarla esperando.

Si es necesario escalar aún más el servicio, se puede recurrir a utilizar varios servidores a la vez. En este caso, todos los servidores sirven el mismo modelo, con un ***load balancer o balanceador de carga*** encargándose de dirigir las *requests* a cada uno de ellos según su carga de trabajo.

### Versionar los modelos

Finalmente, otra característica importante en un servidor es la posibilidad de versionar los modelos disponibles. De este modo, si lo mejoramos entrenándolo con nuevos datos o con una arquitectura distinta, es posible subir el modelo con una nueva versión permitiendo, además, hacer un *rollback* al modelo anterior si el nuevo no está funcionando como esperábamos.

## Servidores de modelos

El uso generalizado del *deep learning* en los últimos años ha hecho surgir una serie de servidores para modelos de *machine learning* que incorporan las características comentadas anteriormente, así como otros aspectos que facilitan la puesta en producción de modelos entrenados con frameworks como TensorFlow, Caffe, Keras, etc. La idea de estos «servidores de modelos» es sencilla: el desarrollador aporta el modelo salvado en disco (por ejemplo, un fichero contenido un modelo en formato de TensorFlow) y el servidor se encarga de responder las *requests* de una manera escalable y eficiente.

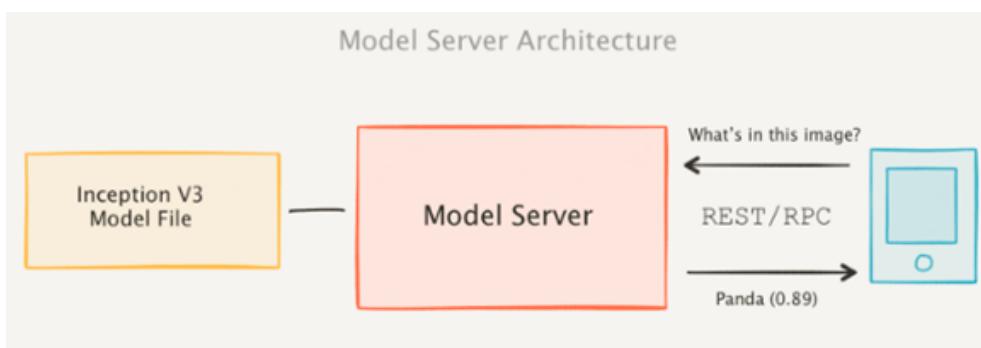


Figura 138. Servidor de modelos.

Fuente: <https://medium.com/@vikati/the-rise-of-the-model-servers-9395522b6c58>

**TensorFlow Serving** es el servidor oficial de TensorFlow. Forma parte del ecosistema de este y, si bien se puede generalizar a otros formatos, aporta grandes facilidades para poner en producción modelos de este framework. Incorpora interfaces gRPC y REST para responder las *requests* y es utilizado en Google para muchos sistemas en producción.

**Clipper** es otro servidor para modelos de *machine learning*, desarrollado en este caso en la universidad de Berkeley. No impone el uso de ningún framework y favorece el desarrollo en *containers*, facilitando la puesta en producción de modelos. La comunicación con el servidor es a partir de una interfaz REST.

Otros servidores disponibles son **DeepDetect** y **Model Server for Apache MXNet**.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 10.3. Ecosistemas en la nube



Accede al vídeo «Ecosistemas en la nube» a través del aula virtual

Antes de la irrupción de los servicios en la nube, cuando una compañía quería poner en marcha un servicio en la red, tenía que adquirir normalmente el hardware necesario para ello y tener una su propia sala de servidores. Las empresas de *hosting* se dedicaban básicamente a servir páginas webs sencillas.

Todo eso cambió en 2006 cuando Amazon presentó **Amazon Web Services (AWS)**. AWS es una plataforma de servicios en la nube que admite el desarrollo de soluciones informáticas. En particular, AWS ofrece capacidad de cómputo (servidores), bases de datos, almacenamiento masivo de datos y una larga lista de herramientas y recursos IT que permiten a una empresa poner en funcionamiento complejos sistemas informáticos sin tener que preocuparse del hardware para llevarlo a cabo.

AWS y otros ecosistemas en la nube han revolucionado el desarrollo de soluciones informáticas. Con la computación en la nube, ya no es necesario instalar y mantener una costosa infraestructura informática, sino que basta con pagar por los recursos que se necesitan bajo demanda, lo que facilita también escalar los servicios informáticos a miles o millones de usuarios de una manera más sencilla. Si necesitamos más servidores para ofrecer un producto, no es necesario tener que instalar un *datacenter* mayor: es suficiente con pagar el uso de nuevas máquinas a nuestro proveedor en la nube.

Los mayores proveedores de computación en la nube en la actualidad son AWS, Google Cloud Platform (GCP) y Microsoft Azure. Como no podía ser de otra manera, ante el ascenso de la inteligencia artificial, estos están empezando a ofrecer un amplio abanico de soluciones de computación para esta área. Una muestra es la posibilidad de contratar máquinas específicas con GPU para el entrenamiento y predicción de modelos. Algunos proveedores como Google ofrecen incluso hardware específico para el entrenamiento de modelos de *deep learning*, como las TPU (*Tensor Processing Unit*).

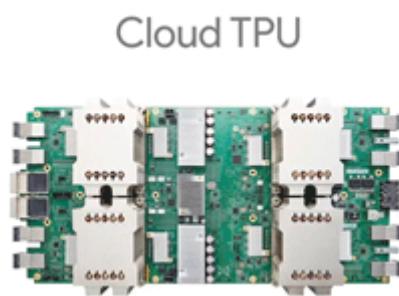


Figura 139. TPU.

Fuente: <https://youtu.be/78P0pBj-i4c>

Del mismo modo, los ecosistemas en la nube ofrecen una serie de **API para resolver tareas comunes** en el mundo del *machine learning*:

- ▶ Procesamiento del lenguaje natural: problemas sobre textos como análisis de sentimiento, extracción de entidades, *POS tagging*...
- ▶ *Speech-to-Text* y *Text-to-Speech*: convertir audio a texto y texto a audio, respectivamente.
- ▶ Traducción automática: traducción de textos.
- ▶ Visión: clasificación de imágenes, detección de objetos y caras en imágenes o extracción de texto desde imágenes, entre otros.
- ▶ Análisis de vídeo: tareas como extracción de contenidos en vídeos y etiquetado.
- ▶ Chatbots: frameworks de *machine learning* para el desarrollo de asistentes conversacionales, especialmente alrededor de la detección de la intención del usuario.

Este tipo de API resuelven problemas muy comunes y para los cuales una solución general puede dar buen resultado. Por ejemplo, la traducción automática es un problema definido de manera muy clara, para el cual se necesita una gran cantidad de datos y una labor compleja de refinamiento de los modelos. Recopilar todos estos datos y entrenar un modelo propio sería demasiado caro y complicado para algunas de estas tareas, por lo que en ocasiones merece la pena recurrir a estas soluciones «*off-the-shelf*».



Accede a los ejercicios de autoevaluación a través del aula virtual

## 10.4. Aspectos prácticos de la puesta en producción de sistemas de *machine learning*



Accede al vídeo «Aspectos prácticos de la puesta en producción de sistemas de *machine learning*» a través del aula virtual

La puesta en producción de un sistema de aprendizaje automático es un proceso muy complejo. De hecho, la parte del sistema que corresponde puramente al código de ML es una pequeña parte de todo el proceso, como se ve en la siguiente imagen:

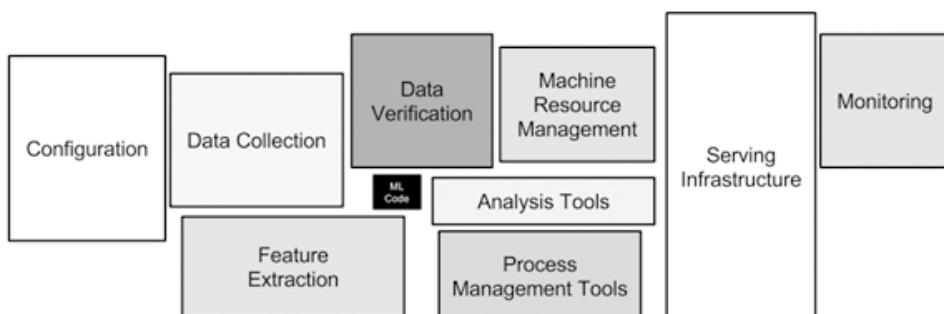


Figura 140. Sistema de aprendizaje automático.

Fuente: <https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf>

En la imagen podemos ver una serie de aspectos necesarios para implementar un sistema de *machine learning*:

- ▶ Una parte importante es la **recolecta** de datos, así como la **extracción** de *features* de estos datos y una posterior **validación** de los mismos.

Antes de implementar nuestros modelos y entrenarlos, es necesario saber qué datos necesitamos y cuáles de ellos tienen sentido para la tarea que queremos resolver. Este proceso puede volverse extraordinariamente complejo.

Por ejemplo, podríamos necesitar interacciones de usuarios de una red social: qué *likes* se han dado, quién es amigo de quién, qué tipos de vídeos y páginas ha visto un usuario, etc. Estos datos pueden estar guardados en formato de *logs* o en distintas bases de datos, de modo que se hace necesario escribir código para extraerlos y tratarlos de modo que sean útiles para nuestro algoritmo. Igualmente, es posible que los datos que necesitamos no estén siendo recopilados hasta el momento, por lo que sería necesario trabajar en conseguir que esos datos se capturen en nuestro sistema.

- ▶ Asimismo, como hemos visto en los apartados anteriores de este tema, es necesario contar con una **infraestructura para entrenar y servir nuestros modelos**, para lo cual podemos utilizar un proveedor de *cloud computing* y un servidor de modelos, como hemos visto en los apartados anteriores.
- ▶ Del mismo modo, es importante contar con una serie de elementos de **monitorización y análisis** del funcionamiento de nuestro sistema. Estos guardan datos acerca de cuántas *requests* se reciben, cuál es el tiempo de respuesta de estas, etc. El sistema de monitorización debería de alertar cuando los servidores están caídos o cuando el tiempo de respuesta es demasiado alto, lo cual nos permite actuar a tiempo para evitar que nuestro sistema deje de funcionar o que los usuarios reciban un servicio lento y con errores.

Finalmente, hay una serie adicional de aspectos prácticos más propios del aprendizaje automático que conviene tener en cuenta y que pueden hacer que nuestro sistema en producción no funcione como esperábamos. Veremos dos de ellos: el *training-serving skew* y el comportamiento del modelo en subpoblaciones.

### *Training-serving skew*

*Training-serving skew* es el problema que viene dado por una **degradación del rendimiento** entre el entrenamiento del modelo y su uso para inferencia. Es muy típico entrenar un modelo y ver que todas nuestras métricas de entrenamiento (métricas *offline* como *accuracy*, *precision*, *recall*, AUC, etc.) son muy buenas, por lo que ponemos a nuestro modelo a funcionar en entornos reales con tranquilidad. Sin embargo, al comprobar con detalle lo que el modelo está haciendo en el sistema en producción, nos damos cuenta de que este no está funcionando tan bien como nos parecía. Hay varias **razones** por las que esto ocurre:

- ▶ Una de ellas es la existencia de una discrepancia entre **cómo se extraen los datos** para entrenar y los datos que el modelo recibe durante *serving*. Es común tener caminos de código distintos para generar estos datos, lo que puede dar lugar a que ciertas *features* sean calculadas de manera un poco distinta o que, en ocasiones, directamente no aparezcan. ¿Se han normalizado los valores de una *feature* de la misma manera a la hora de entrenar que a la hora de servir? ¿Estamos seguros de que la edad del usuario está disponible a la hora de servir, tal y como estaba durante el entrenamiento? El hecho de que nuestro algoritmo reciba una distribución de datos muy diferente con la que fue entrenado, o que incluso algunos datos presentes durante el entrenamiento desaparezcan, puede provocar que el modelo se vuelva inestable y su rendimiento disminuya en gran medida, haciendo incluso que los resultados sean totalmente erróneos.
  - Una solución a este problema consiste en asegurarse de que el código utilizado es el mismo o es totalmente equivalente a la hora de extraer los datos para entrenamiento e inferencia.

- Otra forma de hacer esto es, en el caso de que sea posible, recopilar los datos de entrenamiento para futuros modelos a la hora de servir. Si nuestro sistema guarda los datos que ve durante *serving*, esos valores se pueden utilizar luego para entrenar el modelo y podemos, por tanto, estar seguros de que los datos son los adecuados.
- Otra razón por la que aparece el *training-serving skew* es por utilizar directamente **distribuciones de datos distintas** al entrenar y al servir. Por ejemplo, podemos entrenar un modelo con datos de 2017, pero si lo utilizamos en 2020 es muy posible que los datos que vea este modelo sean de una naturaleza distinta: un sistema de recomendación de películas entrenado en 2017 no sabrá nada acerca de los nuevos estrenos de 2020. En este caso, estamos ante distribuciones de datos distintas, así, nuestro modelo no será capaz de realizar predicciones razonables. De manera similar, si entrenamos un modelo con una subpoblación de usuarios (por ejemplo, usuarios españoles), podríamos encontrarnos con un *training-serving skew* si lo utilizamos con usuarios alemanes, cuya distribución de datos puede presentar diferencias en nuestra aplicación.

## Comportamiento del modelo en subpoblaciones

Otro problema que podemos encontrarnos en la práctica es que un modelo se comporte de manera **muy inestable** para distintos subconjuntos de los datos. Esto puede ser especialmente crítico en el caso de que nuestro modelo trate con usuarios.

### Ejemplo ilustrativo

Imaginemos el siguiente caso. Ponemos en producción un nuevo sistema de recomendación de canciones con buenas métricas de entrenamiento y que también está consiguiendo buenas métricas *online* para nuestros usuarios, esto es, las métricas que tenemos para medir el rendimiento del modelo en producción están siendo buenas, por ejemplo, que el tiempo de escucha está aumentando. Sin embargo, pronto empezamos a recibir

quejas de usuarios acerca de que sus recomendaciones son mucho peores repentinamente.

Puede que las métricas hayan mejorado en general y la mayoría de usuarios estén recibiendo una mejor experiencia, pero a costa de que un pequeño subconjunto de usuarios haya empeorado en gran medida. Estos podrían ser usuarios de cierta zona geográfica, de una edad o género en particular..., incluso podrían ser los usuarios que pagan por el servicio, lo cual sería desastroso para el producto.

Este problema es complicado de tratar y requiere contar con herramientas que nos permitan detectarlo, idealmente, antes de poner el sistema en funcionamiento.

### *Continuous training pipelines*

Otra complicación que surge al poner sistemas de *machine learning* en producción es la periodicidad con la que necesitamos actualizar nuestro modelo. En ocasiones, es suficiente con entrenar un modelo y comprobar que funciona correctamente. Por ejemplo, un clasificador de imágenes de razas de perro puede ser implementado una vez y no necesitar grandes mejoras en el futuro, ya que lo más probable es que las razas de perro no cambien mucho en apariencia en el futuro cercano.

Sin embargo, y relacionado con lo que hemos visto con *training-serving skew*, ciertos datos tienen una **importante componente temporal**. Por ejemplo, un sistema de recomendación de videos y canciones tiene que mantenerse actualizado con los hábitos actuales de los usuarios. O, de manera similar, un clasificador de documentos que detecta *spam* tiene que estar al corriente de los nuevos tipos de *spam* que van apareciendo.

Esto implica que, en muchas ocasiones, no vale con entrenar el modelo una vez y olvidarse del asunto, sino que se necesita recopilar datos actualizados y reentrenar el modelo cada cierto tiempo para que se mantenga actualizado. Llega un momento en el que la periodicidad necesaria para hacer esto (podría llegar a ser semanal o

incluso diaria) implica que este sistema tenga que automatizarse. Esto es lo que se conoce como una *continuous training pipeline* y conlleva un nuevo nivel de complejidad técnica a tener en cuenta, ya que se hace **necesario automatizar** todos el proceso de entrenamiento, evaluación, *deployment* a un servidor...

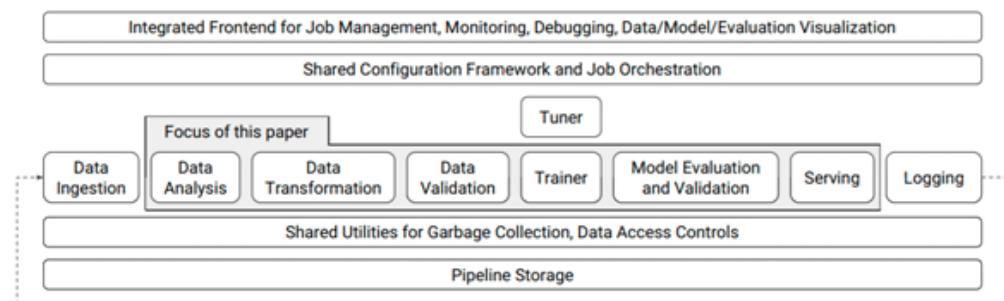


Figura 141. *Continuous training pipeline*.

Fuente: <https://dl.acm.org/citation.cfm?id=3098021>

Una *continuous training pipeline* consta de varias partes, entre las cuales se puede encontrar, dependiendo del caso en particular:

**Ingestión y transformación de datos.** Componente del sistema que se encarga de obtener los datos de diversas procedencias y hacer las transformaciones necesarias para su uso en modelos de *machine learning*.

**Validación de datos.** Se encarga de, una vez tenemos los datos extraídos y listos para usar, realizar comprobaciones para asegurarnos de que esos datos son correctos. Por ejemplo, comprobar que no faltan *features* o asegurarnos de que los datos siguen cierta distribución en consonancia con ejecuciones anteriores de la *training pipeline*. Este componente puede parar la ejecución de la *pipeline* si se detectan anomalías.

**Entrenamiento.** Proceso de entrenamiento del modelo, potencialmente con *hyperparameter tuning* según métricas *offline*.

**Evaluación y validación del modelo.** Componente que se encarga de evaluar y validar el correcto funcionamiento del modelo. Primero, asegurándose de que este puede ser servido y no da lugar a errores de ejecución y, segundo, validando ciertas métricas *offline*. Por ejemplo, asegurándose de que la *accuracy* se mantiene en unos márgenes similares a los de ejecuciones anteriores. En esta parte podría medirse también el comportamiento del modelo en varias subpoblaciones de importancia, deteniendo la ejecución si encontramos anomalías.

**Despliegue del modelo en servidores.** Envío del modelo a los servidores de inferencia y puesta final en producción.

Para una mayor seguridad, una vez el nuevo modelo está en funcionamiento, se podría tener un componente midiendo factores como el *training-serving skew*.



Accede a los ejercicios de autoevaluación a través del aula virtual



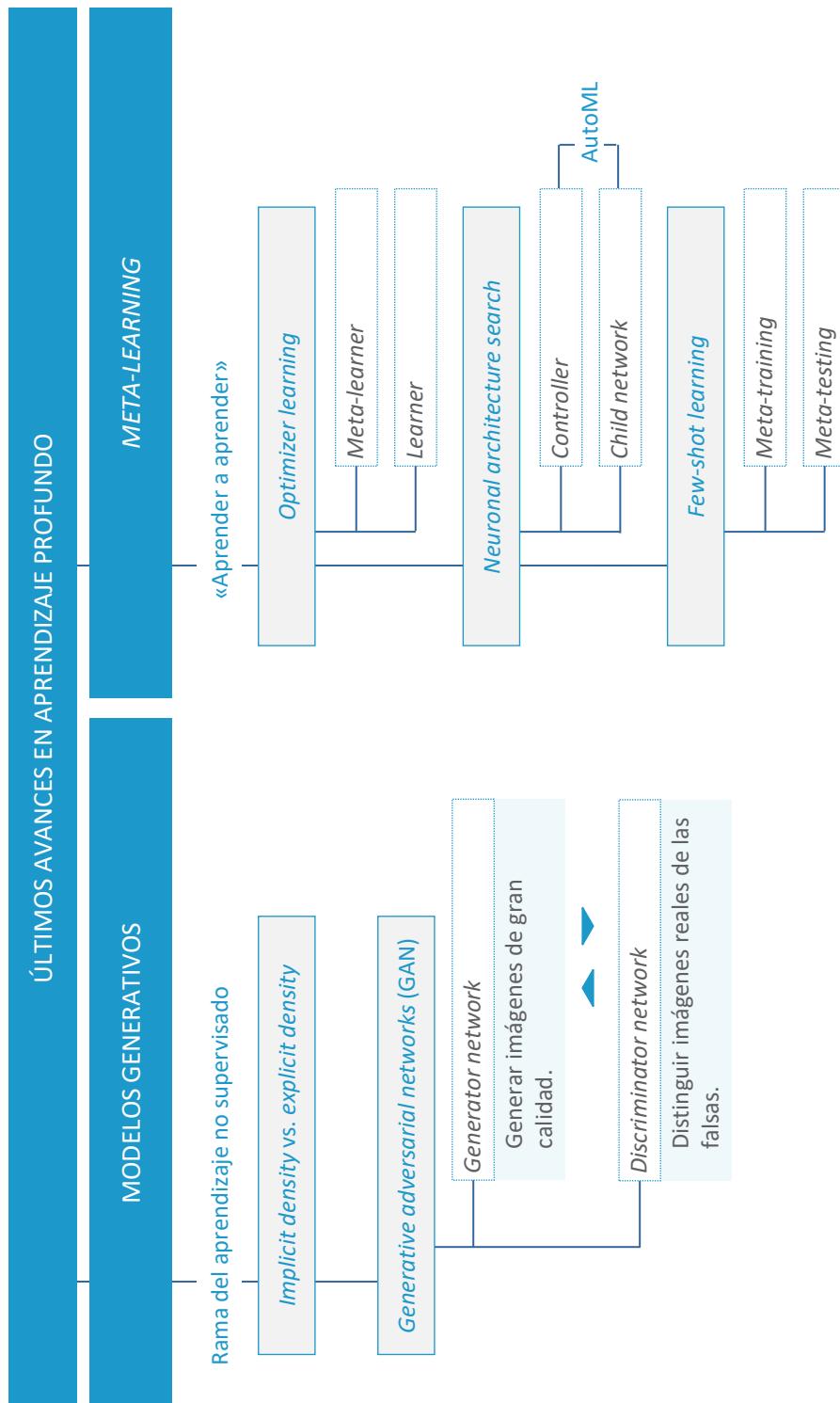
Accede al vídeo «Resumen» a través del aula virtual

# Tema 11. Últimos avances en aprendizaje profundo

## Índice

Esquema	3
Ideas clave	4
11.1. Introducción y objetivos	4
11.2. <i>Generative adversarial networks (GAN)</i>	4
11.3. <i>Meta-learning</i>	8
11.4. Referencias bibliográficas	12

# Esquema



## 11.1. Introducción y objetivos



Accede al vídeo «Introducción y objetivos» a través del aula virtual

En este tema veremos algunos de los últimos avances y líneas de investigación en aprendizaje profundo. En particular, un nuevo tipo de red neuronal, las *generative adversarial networks* (GAN), y el concepto de *meta-learning*.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 11.2. Generative adversarial networks (GAN)



Accede al vídeo «*Generative adversarial networks (GAN)*» a través del aula virtual

Una de las áreas de investigación con más movimiento en la actualidad, dentro del mundo del aprendizaje profundo, es el de las ***generative adversarial networks (GAN)***, introducidas por primera vez por Ian Goodfellow en 2014.

Las GAN son un tipo de **modelo generativo**. Los modelos generativos son una rama del aprendizaje no supervisado en la que, dado un conjunto de datos de entrenamiento, se intenta aprender la distribución de probabilidad de esos datos. A partir de esta distribución de probabilidad, podemos generar nuevos ejemplos similares a los de entrenamiento. Por ejemplo, si dado un conjunto de imágenes, nuestro modelo es capaz de aprender la distribución de probabilidad de esas

imágenes, es posible generar nuevas imágenes parecidas a las originales mediante el muestreo (*sample*) de la distribución aprendida.



Training data  $\sim p_{\text{data}}(x)$



Generated samples  $\sim p_{\text{model}}(x)$

Figura 142. Modelos generativos.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture12.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture12.pdf)

Esto es similar a lo que vimos en los **modelos del lenguaje**. En ellos considerábamos que el texto venía dado por una distribución de probabilidad en la que cada palabra depende de las palabras anteriores. Como vimos, la distribución de probabilidad aprendida puede ser usada para generar texto. En el caso de los modelos del lenguaje, el objetivo era aprender directamente una función de densidad. Esta estrategia para obtener un modelo generativo es usada también en otras técnicas actuales como:

- ▶ PixelRNN.
- ▶ PixelCNN.
- ▶ Variational Autoencoders (VAE).

En las GAN, a diferencia de estos últimos, no se intenta modelar una función de densidad (con la que posteriormente se pueden generar ejemplos), sino que directamente se aprende a generar ejemplos. Este tipo de modelo generativo se conoce como ***implicit density estimation*** (en oposición a ***explicit density estimation***).

La popularidad de las GAN se debe principalmente a sus impresionantes resultados, con modelos capaces de generar imágenes de gran realismo. Se aproximan al problema de la generación desde el punto de vista de un juego para dos jugadores. Estos dos jugadores son dos redes neuronales conocidas como *generator network* y *discriminator network*.

La **generator network** es una aproximación a la obtención de una muestra (*sample*) de la distribución de los datos de entrenamiento. En otras palabras: se encarga de generar imágenes parecidas al *training set*. Su *input* es un vector aleatorio, un vector que podemos cambiar a placer para generar imágenes distintas.

La **discriminator network** es una red que intenta distinguir entre imágenes reales (pertenecientes a los datos de entrenamiento) y falsas (generadas por la *generator network*).

El juego es el siguiente: la *generator network* tiene como objetivo engañar a la *discriminator network* mediante la generación de imágenes que pasen por ser reales a ojos de esta. Por su parte, la *discriminator network* intenta aprender la diferencia entre imágenes reales y falsas para no ser engañada. De este modo, al entrenar una GAN obtenemos una *generator network* que es capaz de producir réplicas de gran calidad, gracias a que intenta engañar a una red discriminadora que, a su vez, se vuelve cada vez más efectiva al ser entrenada. En definitiva, obtenemos una red generadora que ha aproximado el proceso de obtención de muestras a partir de la función de densidad de los datos de entrenamiento, pero sin modelar esta función directamente.

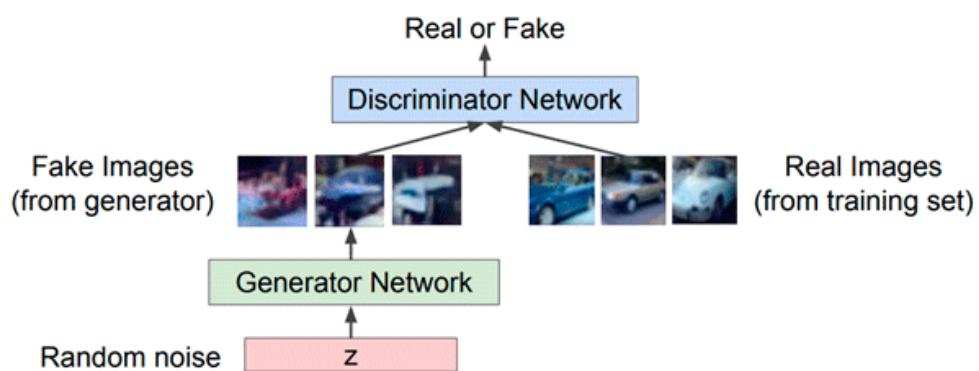


Figura 143. *Generative Adversarial Network*.

Fuente: [http://cs231n.stanford.edu/slides/2018/cs231n\\_2018\\_lecture12.pdf](http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture12.pdf)

Las dos redes son entrenadas a la vez. El entrenamiento o juego entre la *generator* y la *discriminator network* se plasma en la siguiente *objective function*:

$$\min_{\theta_g} \max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log \left( 1 - D_{\theta_d} \left( G_{\theta_g}(z) \right) \right) \right]$$

Donde:

- ▶  $D$  es la función del discriminador, con salida de un valor entre 0 y 1 (la probabilidad de que la imagen sea real).
- ▶  $G$  representa al generador (con salida de una imagen).

De este modo, **el discriminador intenta maximizar el objetivo**, de manera que:

- ▶  $D(x)$  sea lo más cercano posible a 1 (imagen real).
- ▶ Y  $D(G(z))$  sea lo más cercano posible a 0 (*fake*).

Por su parte, **el generador intenta minimizar el objetivo**, de manera que el discriminador piense que las imágenes generadas sean reales, esto es, que  $D(G(z))$  sea cercano a 1.

Como podemos imaginar del hecho de que estemos entrenando dos redes a la vez, el proceso de entrenamiento de una GAN es algo complejo y relativamente inestable, por lo que hay una serie de trucos y mejores prácticas para conseguir entrenarlas con éxito. Esto hace que el entrenamiento de GAN sea un área de investigación muy candente en estos momentos.

El proceso de entrenamiento seguido en el artículo original de Ian Goodfellow (2014) se describe en el siguiente algoritmo.

---

**Algorithm 1** Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator,  $k$ , is a hyperparameter. We used  $k = 1$ , the least expensive option, in our experiments.

---

```
for number of training iterations do
    for  $k$  steps do
        • Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
        • Sample minibatch of  $m$  examples  $\{x^{(1)}, \dots, x^{(m)}\}$  from data generating distribution  $p_{\text{data}}(x)$ .
        • Update the discriminator by ascending its stochastic gradient:
            
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))] .$$

    end for
    • Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_g(z)$ .
    • Update the generator by descending its stochastic gradient:
        
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))) .$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.
```

---

Figura 144. Proceso de entrenamiento seguido por Goodfellow.

Fuente: Goodfellow et al. (2014, p. 4).



Accede a los ejercicios de autoevaluación a través del aula virtual

### 11.3. *Meta-learning*



Accede al vídeo «*Meta-learning*» a través del aula virtual

*Meta-learning* es otra área de investigación de gran interés para la comunidad de inteligencia artificial en la actualidad, si bien sus inicios datan de hace varias décadas.

La idea básica de *meta-learning* es aprender y mejorar el proceso del aprendizaje o, en otras palabras, «aprender a aprender».

Los humanos somos muy versátiles en aprender nuevas tareas recibiendo una mínima información. Por ejemplo, somos capaces de comprender cómo es un tipo de objeto sin necesidad de recibir miles de imágenes de distintas instancias de ese objeto, así como de utilizar nuestras experiencias pasadas para aprender nuevas

tareas. A diferencia de nosotros, los algoritmos que hemos visto hasta ahora suelen ser muy efectivos a la hora de hacer una tarea específica, pero no son capaces de generalizar a nuevas tareas y, además, necesitan una gran cantidad de información específica (pensemos, por ejemplo, en todas las imágenes necesarias para un detector de objetos o en las grandes cantidades de texto para entrenar un modelo del lenguaje).

De este modo, nos gustaría que los sistemas de inteligencia artificial fueran más versátiles y pudieran aprender una larga serie de habilidades, sin que cada habilidad requiriera de un arduo proceso de obtención de datos y de un largo entrenamiento posterior. En definitiva, nos gustaría que los sistemas de inteligencia artificial «aprendieran a aprender» nuevas tareas de manera rápida usando su experiencia de aprendizaje anterior, en vez de considerar cada tarea de manera aislada. Este es el objetivo del *meta-learning*, una compleja área de investigación, pero que está empezando a arrojar resultados prometedores. En este tema veremos algunos acercamientos al problema.

Es importante mencionar que *meta-learning* es un concepto distinto (aunque relacionado) del *hyperparameter tuning*. En este, como sabemos, se intenta encontrar la mejor combinación posible de hiperparámetros de un modelo en particular, de modo que obtengamos los mejores resultados posibles a la hora de entrenar ese modelo.

Veamos ahora algunas formas de definir el problema de *meta-learning*.

### *Optimizer learning*

Como hemos visto en este curso, la elección del optimizador es de gran importancia a la hora de entrenar redes neuronales profundas. Existe una gran variedad de ellos y, pese a que sabemos que algunos suelen funcionar bien, no siempre está claro cuál es la mejor opción para nuestra arquitectura en concreto. De este modo, podríamos preguntarnos: ¿por qué no aprendemos también el optimizador a utilizar, de manera

que sea óptimo para que nuestra red aprenda de manera más rápida la tarea a resolver?

En este tipo de *meta-learning*, hay una red neuronal auxiliar, conocida como *meta-learner*, que se encarga de aprender cómo hacer un *update* de los parámetros de la red principal (*learner*). La red *meta-learner* suele ser normalmente una RNN, lo cual facilita que esta recuerde cómo ha cambiado los parámetros en iteraciones anteriores. Esta red puede ser entrenada mediante aprendizaje supervisado o *reinforcement learning*.

### *Neural architecture search*

Otro acercamiento al problema de *meta-learning* es la *neural architecture search*. Como hemos visto durante el curso, la búsqueda de arquitecturas de redes neuronales es una tarea compleja que requiere un elevado grado de conocimiento y experimentación. Una clara referencia son las complejas redes para *computer vision* que vimos en el tema de *convolutional networks*.

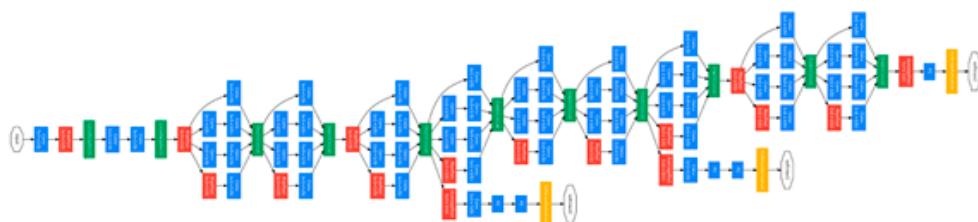


Figura 145. Arquitectura GoogleNet.

Fuente: <https://ai.googleblog.com/2017/05/using-machine-learning-to-explore.html>

*Neural architecture search* trata el problema de **diseñar una red** como un problema de *meta-learning*: se intenta aprender a diseñar redes neuronales de manera que aprendan de manera eficiente distintas tareas.

Un ejemplo de este tipo de aproximamiento al *meta-learning* es AutoML de Google.

En este:

- ▶ Una red neuronal recurrente (*controller*) se encarga de proponer un modelo (*child network*), que es entrenado y evaluado para una tarea.
- ▶ El *feedback* recibido de esta evaluación se utiliza para informar al *controller* (siguiendo un marco de *reinforcement learning*) de cómo han afectado los cambios en la tarea.
- ▶ Este proceso se repite miles de veces, permitiendo al *controller* diseñar modelos de mayor calidad para la tarea en cuestión.

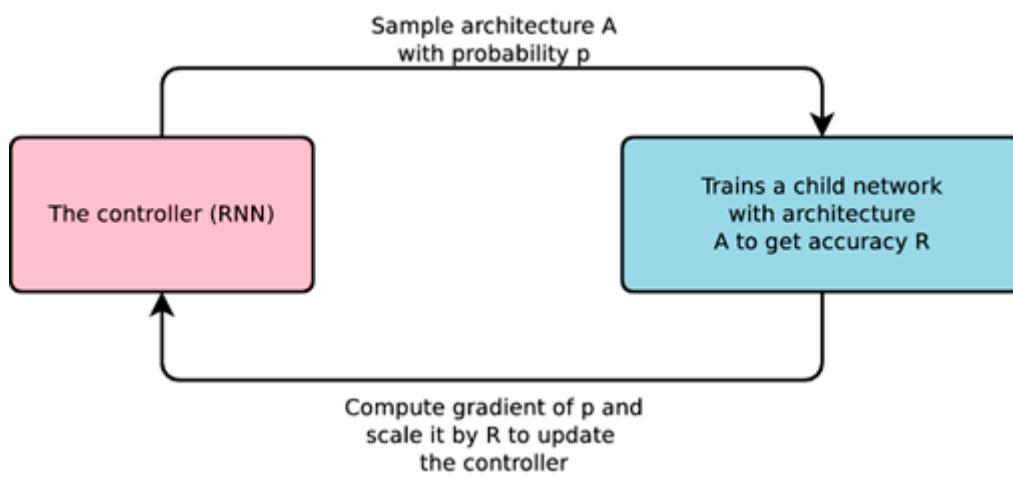


Figura 146. Ejemplo de *Neural Architecture Search*.

Fuente: <https://ai.googleblog.com/2017/05/using-machine-learning-to-explore.html>

### *Few-shot learning*

En *few-shot learning*, el objetivo es conseguir que una máquina pueda generalizar conceptos y aprender a partir de un **número reducido de ejemplos**, tal y como haría un humano, y no a partir de datasets de gran tamaño. El problema se suele formular de manera que tenemos una serie de tareas distintas a aprender (cada una con sus correspondientes datos de entrenamiento y test) y queremos evaluar la capacidad de nuestro sistema de *meta-learning* para aprender nuevas tareas.

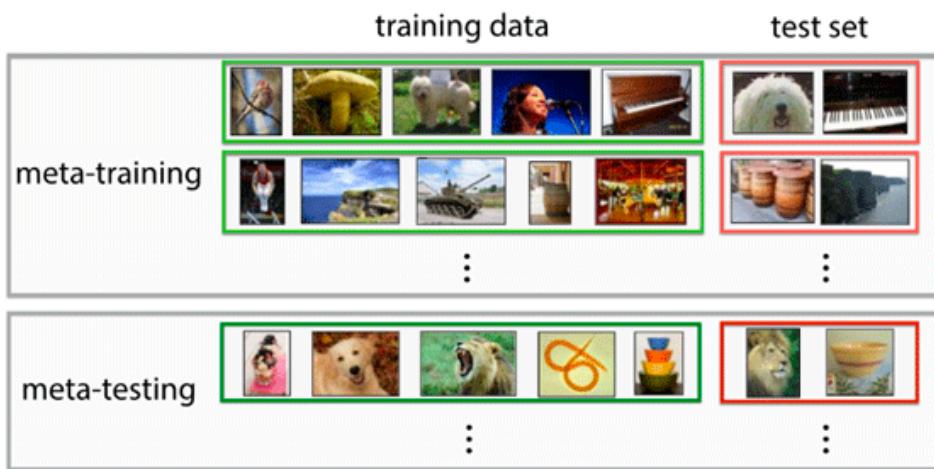


Figura 147. Ejemplo de *few-shot learning*.

Fuente: <https://bair.berkeley.edu/blog/2017/07/18/learning-to-learn/>

En la imagen anterior, vemos un ejemplo con tareas de clasificación de imágenes (cada fila de imágenes es una tarea distinta). Durante el proceso de *meta-learning*, el modelo, (o más bien, meta-modelo), es entrenado para aprender tareas en el set de *meta-training*, y su capacidad de aprender nuevas tareas es evaluada en el set de *meta-testing*.

Para cada tarea se entrena un modelo diferente que se encarga de resolver dicha tarea. Por lo tanto, se manejan **dos niveles de optimización** a la vez: el *learner*, que aprende las tareas a realizar, y el *meta-learner*, que entrena al primero de modo que este sea efectivo aprendiendo a partir de la limitada cantidad de datos disponible.



Accede a los ejercicios de autoevaluación a través del aula virtual

## 11.4. Referencias bibliográficas

Goodfellow et al. (2014). Generative adversarial networks. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence y K. Q. Weinberger (Eds.). *Advances in Neural Information Processing Systems 27 (NIPS 2014)*. <https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>



Accede al vídeo «Resumen» a través del aula virtual