ADVERTENCIA PARA NUEVOS LECTORES: En realidad este libro fue escrito hace un año, durante las vacaciones de Navidad de 2015. TensorFlow me cautivó y quería compartir este conocimiento con mis estudiantes y colegas de investigación. Por tanto le pido al lector que tenga en cuenta al leer este libro que se basa en una versión antigua de TensorFlow (tensorflow-0.5.0) y aunque los contenidos son todavía válidos, en estos momentos el libro sólo es adecuado para fines de introducción en el tema. La versión actual (tensorflow-1.0.0-alpha) tiene muchas características nuevas e importantes que no contempla este libro . Honestamente nunca llegué a imaginarme que este libro iba a tener tanta repercusión y tan larga vida. Vivirlo ha sido un honor para mí, gracias a todos y a todas!

ACTUALIZACIÓN (04/05/2017): Github con el código actualizado a la versión TensorFlow 1.1

ACTUALIZACIÓN (04/06/2017): Transparencias del tema actualizadas

Hello World en TensorFlow

Para iniciarse en la programación del Deep Learning

> *Este libro está dedicado a la comunidad de código abierto, cuyo trabajo consumimos todos los días sin saberlo.*

COMPRAR:

versión papel en Amazon/Lulu
versión PDF en Lulu

CONTENIDO:

Fotos presentación libro

Interview about the book

Tags

Artificial Intelligence
Artificial Intelligence
Barcelona
Big Data
Big Data
Big Data Analytics
BSC
BSC
CaixaBank
Cloud Computing
Cloud Computing
cognitive computing
Deep Learning
Deep Learning
emprendedores
entrevista
Europa
FIB
google
Green Computing
Hadoop
High-Performance Big-Data Analytics
High Performance Computing
IEEE
Inteligencia Artificial
Internet of Things
La Vanguardia
La Vanguardia
Libro Empresas en la Nube
Machine Learning
MapReduce
Marenostrum
Master Courses
Mateo Valero
meetup
Predicciones y Tendencias
Python
RAC1
Redes Sociales
Smart Computing
Spark
Supercomputing
TensorFlow
Tutorial
UPC

Categories

› Artificial Intelligence
› Big Data
› Cloud Computing
› Cognitive Computing
› Deep Leaning ES

Prólogo

El área de aprendizaje automático (o *Machine Learning*) ha exhibido una gran expansión gracias al codesarrollo de tecnologías clave, tales como la computación, el almacenamiento de datos masivo e Internet. Muchas de las tecnologías y eventos del día a día de muchas personas están, directa o indirectamente, influenciadas por el aprendizaje automático. Ejemplos tecnológicos como el reconocimiento del habla, la clasificación de imágenes en nuestros teléfonos o la detección de spam en correos electrónicos, han permitido desarrollos de *apps* que, hace apenas una década, sonarían a ciencia ficción. El uso del aprendizaje en modelos bursátiles, médicos, o bolsas de trabajo ha impactado a la sociedad de forma masiva. Además, coches sin conductor, drones o robots de todo tipo van a impactar a la sociedad en un futuro no muy lejano.

El *Deep Learning*, un subcampo de *Machine Learning*, ha sido sin lugar a dudas uno de los campos con expansión explosiva desde que fue redescubierto en el 2006. Efectivamente, una gran parte de los *startups* en Silicon Valley se especializan en ello, y las grandes compañías tecnológicas como Google, Facebook, Microsoft o IBM tienen equipos tanto de desarrollo como de investigación. *Deep Learning* ha generado interés incluso fuera de los ámbitos universitarios y de investigación: numerosas revistas especializadas (como *Wired*) o generales (como *New York Times*, *Bloomberg* o *BBC*) han escrito múltiples artículos sobre el tema.

Este interés ha hecho que muchos estudiantes, emprendedores e inversores se hayan sumado al *Deep Learning*. Gracias a todo el interés generado, varias librerías de desarrollo de modelos se han abierto como "*Open Source*". Siendo yo uno de los principales promotores de la librería que desarrollamos en Berkeley (Caffe) en 2012 como estudiante de doctorado, puedo decir que TensorFlow, presentado en este libro, y diseñado también por el grupo en Google (California), donde he estado investigando desde 2013, va a ser una de las principales herramientas con la que investigadores, pequeñas y grandes compañías van a desarrollar sus ideas alrededor de *Deep Learning* y *Machine Learning*. Garantía de ello es la cantidad de ingenieros e investigadores de primera clase que han participado en este proyecto, que ha culminado con el su *Open Sourcing*.

Espero que este libro introductorio ayude al navegante interesado a empezar su aventura en este campo tan interesante. Y quisiera agradecer a su autor, a quien tengo el placer de conocer personalmente, por el esfuerzo de divulgación de esta tecnología escribiendo este libro en tiempo récord, dos meses desde que se hizo público el proyecto en *open source*. Una muestra más de la vitalidad que tiene Barcelona y de su interés para ser uno de los actores en este escenario tecnológico que tanto va a impactar nuestro futuro.

Oriol Vinyals, Research Scientist en Google Brain

[volver al índice de contenidos]

Prefacio

Education is the most powerful weapon which you can use to change the world.
Nelson Mandela

Nos encontramos ante un reto y una gran oportunidad para los ingenieros e ingenieras que dejaron las aulas universitarias hace tiempo, pero que su día a día les está llevando inexorablemente a tener que aprender potentes y nuevas herramientas que los avances en áreas como *Machine Learning* les están ofreciendo para crear empresas competitivas.

La vertiginosa revolución tecnológica en que nos encontramos inmersos genera que los conocimientos de los trabajadores de las empresas modernas se encuentren en un proceso de transformación constante.

Ejemplo de ello son las grandes empresas, que disponen de ingentes volúmenes de datos que requieren profesionales que sepan de *Machine Learning* para poder sacar valor de estos datos. Y no solo ellas precisan de profesionales con estos conocimientos, ya que esta necesidad se está extendiendo a empresas de todo tipo y tamaño, requiriéndose muchísimos más profesionales que los que las universidades estamos formando en estos momentos.

Este es el caso de un antiguo alumno, y gran amigo, que en estos momentos está convirtiéndose un experto en la materia al estar convencido que su startup Undertile tiene una gran oportunidad

› Deep Learning
› DLAI
› General
› High-Performance Big-Data Analytics
› Inteligencia Artificial
› Internet of Things
› Keras
› La Vanguardia
› Marenostrum
› Personal
› Supercomputing
› Technology

Contact Info

UPC Campus Nord , C6-217
C/ Jordi Girona 1-3
Barcelona 08034

Blog Archives

› November 2017 (1)
› October 2017 (3)
› September 2017 (7)
› August 2017 (1)
› July 2017 (3)
› June 2017 (6)
› May 2017 (2)
› April 2017 (1)
› March 2017 (2)
› February 2017 (2)
› January 2017 (1)
› December 2016 (4)
› November 2016 (3)
› June 2016 (1)
› May 2016 (8)
› April 2016 (5)

aplicando las nuevas tecnologías de *Deep Learning*. Él influyó mucho en mi decisión de escribir este libro. Al hablarle de esta nueva tecnología llamada TensorFlow, que había sido liberada pocos días antes, le brillaron los ojos de la misma forma que los de mis alumnos cuando, en una larga práctica tecla en mano, ven cómo sale aquel resultado tan esperado en su pantalla. Es un estímulo pensar que no solo mis alumnos requieren estar al día de lo último en tecnología, sino que también hay empresas innovadoras que pueden y quieren sacarle provecho.

Por todo ello, el propósito de este libro es el de ayudar a divulgar este conocimiento entre los ingenieros e ingenieras que quieran ampliar sus conocimientos en el apasionante mundo del *Machine Learning*. Creo que cualquiera con una formación en ingeniería puede requerir a partir de ahora conocimientos en *Deep Learning*, y *Machine Learning* en general, para aplicarlo en su actual o futuro trabajo.

El lector se preguntará cómo es que un ingeniero de datos, que en realidad lo es de computadores, se propone este reto de escribir como si fuera un científico de datos. Precisamente por ser ingeniero creo que puedo aportar ese enfoque introductorio al tema, y que esto puede ser de gran ayuda para muchos ingenieros e ingenieras en sus primeros pasos; luego estará en sus manos profundizar en aquello que más le conviene.

Espero con este libro aportar mi granito de arena en este mundo de la formación que tanto me apasiona. El conocimiento debe estar al alcance de todos, y por este motivo el contenido de este libro está disponible en la página web *www.JordiTorres.eu/TensorFlow* totalmente gratuito en formato html. Si al lector le es útil este material y cree oportuno compensar el esfuerzo del autor al escribir el libro, hay una pestaña en la página web para poder hacer una donación.

Por otro lado, si el lector prefiere optar por un ejemplar en papel, podrá adquirir el libro a través de los portales *Amazon.es* y *LuLu.com*. En este caso, las ganancias de las edición en papel serán donadas al proyecto conjunto que están realizando la *startup* UnderTile y la *Fundació El Maresme*, una entidad sin ánimo de lucro, nacida hace cincuenta años, que impulsa la integración social y la mejora de la calidad de vida de las personas con discapacidad intelectual. ¡Muchas gracias por sumaros al proyecto!

Esta idea salió de la misma *startup* Undertile, que juntamente con *Fundació El Maresme* están llevando a cabo un proyecto en el que se están involucrando personas con discapacidad intelectual para recuperar, perdurar y compartir la memoria de las personas a través de la digitalización y etiquetado de las fotos en papel, diapositivas, etc. En la página web también habrá una segunda pestaña para hacer donaciones al proyecto si fuera del interés del lector.

Déjenme decirles ¡gracias por estar leyendo este libro! El simple hecho me reconforta y justifica mi esfuerzo en escribirlo. Aquellos que me conocen saben que la divulgación tecnológica es una de mis pasiones, que me mantienen con vigor y energía. ¡Ah!; para aquellos que sospechan que en realidad el propósito de esta obra era tener una excusa para ilustrar un libro técnico con mis dibujos, otro de me mis *hobbies*, pues están en lo cierto, porqué lo voy a negar; hasta ahora solo había ilustrado libros no técnicos.

Para acabar estas líneas de apertura, me queda tan solo añadir que si el lector requiere contactar conmigo en relación al contenido de este libro, no dude en hacerlo a través del correo electrónico *LibroTensorFlow@gmail.com*. Estaré encantado de contestarle.

Jordi Torres, enero 2016

[volver al índice de contenidos]

Enfoque del libro

Tell me and I forget. Teach me and I remember. Involve me and I learn.
Benjamin Franklin

Una de las aplicaciones habituales del *Deep Learning* incluye el reconocimiento de patrones. Por ello, de la misma manera que cuando uno empieza a programar existe la tradición de empezar por un *print "Hello World"*, en *Deep Learning* se crea un modelo de reconocimiento de números escritos a mano, siendo de uso habitual los contenidos en el dataset MNIST[1], muy conocido en el

› March 2016 (11)
› February 2016 (4)
› January 2016 (4)
› December 2015 (4)
› November 2015 (8)
› October 2015 (6)
› September 2015 (4)
› August 2015 (1)
› July 2015 (5)
› June 2015 (4)
› May 2015 (5)
› April 2015 (8)
› March 2015 (7)
› February 2015 (1)
› January 2015 (6)
› December 2014 (5)
› November 2014 (6)
› October 2014 (4)
› September 2014 (2)
› July 2014 (2)
› June 2014 (9)
› May 2014 (7)
› April 2014 (5)
› March 2014 (4)
› February 2014 (4)
› January 2014 (3)
› December 2013 (6)
› November 2013 (4)
› October 2013 (2)
› August 2013 (1)
› July 2013 (5)
› May 2013 (1)
› April 2013 (2)
› March 2013 (2)

área. Este va a ser el ejemplo principal del libro, que con diversas aproximaciones me permite acercar al lector a esta nueva tecnología llamada TensorFlow.

No obstante, no pretendo aquí escribir un tratado sobre *Machine Learning* ni *Deep Learning*, y por ello pido de antemano disculpas a mis colegas científicos de datos por ciertas simplificaciones que me he permitido en aras de llegar a un lector profano a este conocimiento, que encontrará aquí un formato habitual al que uso en mis clases; se trata de invitarles a usar el teclado de su ordenador mientras va aprendiendo. Nosotros le llamamos "*learn by doing*", y mi experiencia como profesor en la UPC me indica que es una aproximación que funciona muy bien en ingenieros e ingenieras que tratan de iniciarse en un nuevo tema.

Por esta razón el libro tiene un carácter eminentemente práctico, y por ello se ha reducido todo lo posible la parte teórica, aunque es estrictamente necesario recurrir a ciertos detalles matemáticos, cosa que hará que, en las siguientes páginas, se vayan intercalando conocimientos teóricos y prácticos de forma que se complementen en el proceso de aprendizaje.

La parte práctica, como es de esperar, estará basada en TensorFlow, el paquete de código abierto que recientemente ha liberado Google. Conocer TensorFlow creo que va a ser importante en el currículum de cualquier ingeniero o ingeniera en los próximos meses.

En el primer capítulo, además de hacer una introducción al escenario, donde, a mi parecer, TensorFlow tendrá un papel relevante, aprovecho para dar un primer vistazo a la estructura que presenta un código TensorFlow explicando brevemente los datos que este mantiene internamente. Como el lector podrá observar, iremos avanzando en paralelo los ejemplos prácticos con la introducción de aprendizaje de TensorFlow y/o conocimientos requeridos de *Machine Learning*.

En el segundo capítulo, a través de un ejemplo de regresión lineal, presentaré los diferentes elementos importantes en un proceso de aprendizaje, como son la función de coste o el algoritmo *gradient descent*, y, al mismo tiempo, empezar a ver cómo son los datos básicos que maneja TensorFlow.

En el tercer capítulo, a propósito de presentar un algoritmo de clusterización, entraré en detalle a presentar la estructura de datos básica de TensorFlow, llamada tensor, y las diferentes clases y funciones que el paquete TensorFlow ofrece para crearlos y manejarlos.

En el cuarto capítulo, una vez ya introducido en capítulos previos varios conceptos importantes, se presenta en detalle, paso a paso, cómo se puede construir una red neuronal de una sola capa para reconocer dígitos escritos a mano. Esto nos permitirá poner en orden todos los conceptos presentados previamente, así como también ver todo el proceso completo de creación y evaluación de un modelo.

El siguiente capítulo parte de la red neuronal vista en el anterior, y presenta cómo se puede construir una red neuronal de varias capas para poder conseguir un mejor resultado en el reconocimiento de los dígitos escritos a mano. Se presentará con más detalle lo que se conoce como redes neuronales convolucionales.

En el sexto capítulo se entra en un tema más específico y probablemente no del interés de todos los lectores, pero no por ello menos importante, como es el aprovechar la potencia de cálculo que presentan hoy en día las GPUs en TensorFlow. Como se introduce en el capítulo 1, las GPUs juegan un rol importante en el proceso de entrenamiento de las redes neuronales.

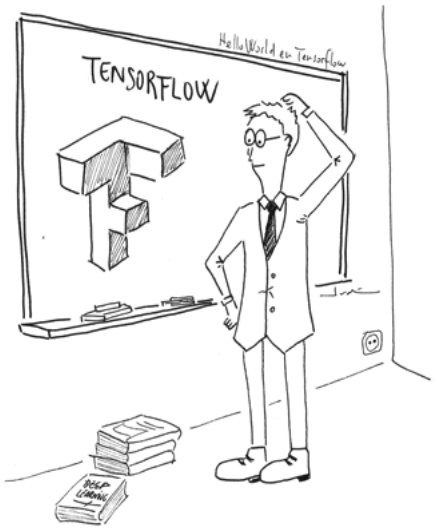
El libro acaba con un capítulo más personal, en el cual me he permitido una pequeña reflexión sobre el tema.

Quisiera recalcar que el código de los ejemplos que aparecen en este libro pueden descargarse en el repositorio *github* del libro[2], y también podrá encontrar las correcciones a las fe de erratas que puedan aparecer en el libro.

Este libro (versión digital) se encuentra en revisión constante. Agradeceré que se notifique a través de LibroTensorFlow@gmail.com cualquier error tipográfico, bug de código o de otro tipo que el lector pudiera encontrar. El lector sería añadido a la lista de agradecimientos en futuras versiones del libro.

[volver al índice de contenidos]

1. Una tecnología al alcance de todos



En este apartado vamos a describir muy brevemente el entorno en el que se enmarca este libro. Intencionadamente, se ha hecho esta introducción teórica breve porque, como se ha dicho en el prefacio se, trata de una obra con carácter eminentemente práctico. Al final de este capítulo se espera que el lector pueda haber instalado el paquete TensorFlow en su ordenador y haber probado un primer ejemplo. También se aprovecha para introducir cómo se estructura habitualmente un código en TensorFlow, especificando primeramente todo el flujo de ejecución y dejando para la última parte del código la ejecución.

Código abierto

El aprendizaje automático (*Machine Learning*) ha sido materia de investigación en la academia durante décadas, pero desde hace pocos años estamos viendo como se incrementa su penetración también en las corporaciones, gracias al gran volumen de datos que estas disponen y a una capacidad de computación, nunca antes vista, puesta a su disposición.

En este escenario, no hay duda que Google, bajo el *holding* de Alphabet, es una de las grandes corporaciones donde las tecnologías de *Machine Learning* han pasado a tomar un protagonismo fundamental en la práctica totalidad de sus iniciativas y productos.

El pasado mes de octubre, cuando Alphabet anunciaba los resultados trimestrales de Google, con subidas considerables en facturación y beneficios, su *CEO* Sundar Pichai decía claramente: "*El* Machine Learning *es la vía de transformación principal que nos está llevando a repensar todo lo que hacemos*".

Tecnológicamente hablando, estamos ante un cambio de era en el que no solo Google es uno de los grandes actores, sino también otras empresas tecnológicas como Microsoft, Facebook, Amazon o Apple, entre muchas otras corporaciones, están también incrementando su inversión en estas áreas.

Hace unas semanas Google ofreció a la comunidad, en forma de código abierto, su motor TensorFlow, una librería de *Machine Learning* con licencia Apache 2.0. TensorFlow puede ser usado por desarrolladores e investigadores que quieren incorporar *Machine Learning* en sus proyectos y productos, de la misma manera que están haciendo internamente en Google con diferentes productos comerciales como son Gmail, Google Photos, Búsquedas, reconocimiento de voz, etc.

Como soy un ingeniero que me dirijo a ingenieros e ingenieras, en el libro miraremos debajo del capó para ver cómo se representa el algoritmo con un grafo de flujo de datos. Esto le permite a TensorFlow poder sacar provecho tanto de CPUs como GPUs en estos momentos, desde plataformas Linux de 64 bits como Mac OS X, así como también plataformas móviles como Android o iOS.

Otro punto fuerte de este nuevo paquete disponible es su visualizador TensorBoard, que permite monitorizar y visualizar mucha información de cómo está funcionando el algoritmo. Poder medir y visualizar resulta sumamente importante en el proceso de creación de mejores modelos. Tengo la sensación que actualmente muchos modelos se afinan a través del proceso un poco a ciegas, mediante prueba y error, con el evidente despilfarro de recursos y, sobretodo, tiempo.

Por tanto, creo que el disponer de TensorBoard es un gran paso en la dirección correcta, y espero que esto anime a la comunidad de *Machine Learning* a validar mejor internamente sus modelos y nuevas prácticas de entrenarlos e inspeccionar su rendimiento. Creo que queda bastante por mejorar en este punto, y aquí los ingenieros e ingenieras podemos aportar mucho.

Permítanme recalcar otra vez que, en especial, TensorFlow nos permite a ingenieros e ingenieras el construir, entrenar y ejecutar redes neuronales *Deep Learning* de manera muy ágil, y por ello lo

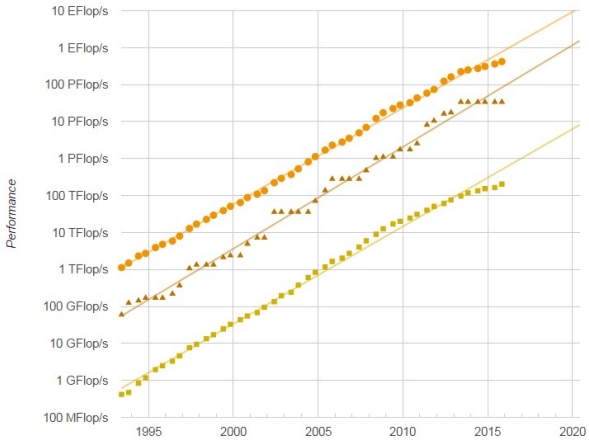
› June 2009 (2)
› May 2009 (1)
› April 2009 (2)
› February 2009 (1)
› November 2008 (3)
› October 2008 (1)
› July 2008 (3)
› June 2008 (2)
› May 2008 (1)
› April 2008 (2)
› February 2008 (1)
› December 2007 (1)
› July 2007 (2)
› June 2007 (1)
› May 2007 (2)
› April 2007 (1)
› February 2007 (1)
› January 2007 (1)
› October 2006 (1)
› July 2006 (1)
› May 2006 (1)

usaremos en este libro. De todas formas, en estos momentos hay una cincuentena de entornos similares para desarrollar algoritmos *Deep Learning* como son Theano, Torch o Caffe, por mencionar algunos. El lector puede encontrar una extensa lista de ellos en la página *Awesome Deep Learning*[3].

Por qué ahora

Sin duda, hasta hace pocos años las técnicas de *Deep Learning* no han experimentado su gran auge, a pesar que están basadas en redes neuronales artificiales que ya aparecían en la literatura científica de los años 80 (por ejemplo con el Neocognitron[4] de Kunihiko Fukushima). Como dato, mencionar que en 1990, Yann LeCun, junto con otros investigadores, consiguieron aplicar el algoritmo estándar *backpropagation* (algoritmo creado a mediados de los 70) a una red neuronal con varias capas con el propósito de reconocer los dígitos de los códigos postales de correo escritos a mano. A pesar del éxito de aplicación del algoritmo, el tiempo requerido para entrenar la red neuronal con ese conjunto de datos fue aproximadamente de 3 días, hecho que lo hacía impracticable para un uso generalizado.

El lector a estas alturas ya debe suponer qué hay de nuevo en el escenario de hoy en día para hacer posible la llegada del *Deep Learning*: una capacidad de computación nunca antes vista. Desde entonces la capacidad de cálculo de los computadores ha experimentado un crecimiento espectacular. Una muestra de ello la podemos encontrar en la lista Top500[5] con los quinientos ordenadores más rápidos del planeta, que se elabora dos veces al año desde 1993. La gráfica resumen de las capacidades de cálculo a que llegan estos ordenadores del Top500, llamados supercomputadores, se muestra a continuación[6]:



En esta gráfica se indica dos veces al año (junio y noviembre) la velocidad máxima de procesado que consigue un supercomputador (en número de instrucciones en coma flotante que un ordenador puede ejecutar por segundo), indicando el más rápido con un triángulo, el ordenador número 500 con un cuadrado y la suma de todos ellos con un círculo. Tenemos datos desde 1993 hasta 2015, siendo en noviembre del 2015 el más rápido del mundo, por sexta vez consecutiva, el supercomputador Tianhe-2, desarrollado por la National University of Defense Technology de China, que llega a una velocidad de 33.86 PetaFlop/segundo, es decir, realizar 33.860.000.000.000.000 operaciones por segundo. ¿Qué les parece?

Como podemos observar en la gráfica, desde que Yann LeCun creó su red neuronal para interpretar códigos postales a hoy en día, la potencia de los ordenadores se ha multiplicado por 1.000.000.000 aproximadamente. Nada despreciable, ¿verdad? Otro tema es que seamos capaces de usar toda esta potencia de cálculo disponible para poder entrenar una red neuronal. Como sufren mis colegas teóricos, de momento queda bastante margen para mejorar. Sin duda, hace falta también el conocimiento de la ingeniería informática en este nuevo mundo del aprendizaje automático. Más adelante ya hablaremos sobre los sistemas de computación basados en arquitecturas especiales como GPUs, que presentan ventajas interesantes para entrenar redes neuronales.

Pero aunque yo me dedique a la supercomputación, y para justificar mi trabajo les intente convencer que este es fundamental para el *Deep Learning*, hay otro factor quizás aun mucho más fundamental por el que podemos asegurar que ahora el *Deep Learning* ha llegado para quedarse. En realidad, para que estos algoritmos sean realmente útiles, se necesitan cantidades ingentes de datos para poder ser entrenados correctamente y hasta ahora los datos disponibles eran muy

limitados. Con la llegada del Big Data en todas sus formas, la generación de datos nunca había sido tan enorme. Como muestra me limito a reproducir unos datos de un interesante artículo en el portal UNIVERSIA[7] que comenta algunas cifras que Bernard Marr[8] presenta en la revista Forbes.com:

- Se han creado más datos en los últimos 2 años que en toda la historia de la humanidad.
- Se estima que para 2020 cada individuo creará 1,7 megabytes de información nueva por segundo.
- Para 2020, nuestro universo de datos pasará de los 4.4 zettabytes que existen actualmente a 44 zettabytes (44 billones de gigabytes).
- Creamos nuevos datos a cada segundo: realizamos 40.000 nuevas búsquedas en Google por segundo, lo que equivale a 3.5 millones de búsquedas por día y 1.2 billones por año.
- En agosto de 2015, se conectaron más de 1.000 millones de personas a Facebook por día, lo que supone un aumento del 3,37% con respecto al mismo período en 2014.
- Los usuarios de Facebook envían una media de 31,25 millones de mensajes y miran 2,77 millones de vídeos por minuto.
- Se sube una media de 300 horas de vídeos a YouTube por minuto.
- En 2015 se tomaron 1 billón de fotografías, y miles de millones serán compartidas en la red. Para 2017 se estima que casi el 80% de las fotografías serán tomadas por *smartphones*.
- Este año, se importarán más de 1.400 millones de *smartphones*, todos ellos con sensores capaces de recolectar todo tipo de datos, sin mencionar aquellos creados por los propios usuarios del móvil.
- Para 2020 tendremos más de 6.100 millones de usuarios de *smartphones* en el mundo.
- En los próximos 5 años habrá más de 50.000 millones de dispositivos *smart* conectados en el mundo, todos desarrollados para recolectar, analizar y compartir datos.

Finalmente, creo que es relevante para el lector hacerle notar que Google en realidad no ha liberado ninguno de sus modelos ya entrenados. Se requieren muchísimos datos y mucha capacidad de computación para entrenar un modelo *Deep Learning* y, al final, un modelo *Deep Learning* ya entrenado es lo que permite las capacidades mostradas por Google de, por ejemplo, su reconocimiento de patrones. Para ser capaz de crear algo comparable, se requiere tener los datos que tiene Google y también sus recursos de computación. Esto es algo que la mayoría de empresas no tienen, y donde continua dejando a Google en una posición de supremacía a pesar de liberar este código.

Machine Learning

Machine Learning es una disciplina científica que trata de que los sistemas aprendan automáticamente. De manera informal, podríamos decir que se trata de un campo de investigación que da a los computadores la habilidad de aprender sin haber sido explícitamente programados para ello, dando a los ordenadores la habilidad de aprender a partir de ejemplos.

Para contentar a mis colegas científicos de datos, reproduzco esta descripción, un poco más formal, de Tom M. Mitchell[9] que dice algo así como: *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."* Por ejemplo, si lo aplicamos al juego del ajedrez, E sería la experiencia de haber jugado muchas veces, T el jugar al ajedrez y P la probabilidad que el programa ganara la siguiente partida.

Los algoritmos de *Machine Learning* se pueden agrupar en diferentes maneras, y como no es el objetivo de este libro avanzar en el conocimiento del *Machine Learning*, sino iniciar al lector en el uso de TensorFlow, podemos resumir que, a grandes rasgos, los algoritmes de *Machine Learning* se clasifican en algoritmos de aprendizaje supervisado y algoritmos de aprendizaje no supervisado. Sin poder entrar en detalle, podríamos decir que en el aprendizaje supervisado, estamos dando al algoritmo un conjunto de datos del que sabemos como debe ser la salida, y donde hay una relación entre la entrada y la salida. A su vez, los problemas de aprendizaje supervisado son clasificados entre problemas de "regresión" y "clasificación".

En un problema de regresión se intenta predecir un resultado dentro de un espacio de salida en forma de función contínua. En cambio, en un problema de clasificación estamos intentando predecir el resultado en un espacio discreto de salida. En el próximo capítulo, usaré un algoritmo de regresión para introducir al lector en la programación en TensorFlow.

Por otro lado, en el aprendizaje no supervisado el proceso se lleva a cabo teniendo información del conjunto de datos formados tan solo por la entrada. No se tiene información sobre que pinta tiene

nuestro espacio de salida en el que se deberá mapear el resultado, por ejemplo que categorías hay de datos. En aprendizaje no supervisado podemos buscar la estructura de los datos, mediante la agrupación (*clustering*) de los datos basándonos en la relación entre las variables de entrada.

En este libro, después de introducir al lector en la programación en Tensorflow con la excusa de un algoritmo de regresión lineal, usaremos un algoritmo de *clustering* para aprender a usar la estructura de datos básica, llamada *tensor*, que maneja TensorFlow.

He creído que organizar el aprendizaje gradual de TensorFlow usando estos métodos básicos en *Machine Learning* (y alguno de sus algoritmos más populares) puede ayudar al lector a avanzar ordenadamente en su aprendizaje a través de este libro, que insisto que su objetivo no es aprender nuevos algoritmos de *Machine Learning,* sino iniciarse en el manejo de TensorFlow.

Instalación de TensorFlow

Ha llegado el momento de pasar el control a nuestro teclado. Les recomiendo que a partir de este punto del libro compaginen su lectura con la práctica en su ordenador.

TensorFlow tiene una API Python (además de una en C/C++) que requiere tener Python 2.7 instalado (doy por sentado que el ingeniero o ingeniera que lea este libro ya lo tendrá instalado o sabrá cómo hacerlo).

En general, cuando se trabaja en Python se recomienda usar el entorno virtual Virtualenv. Virtualenv es una herramienta para mantener las dependencias requeridas en diferentes proyectos Python en diferentes lugares del mismo ordenador. Si usamos Virtualenv para instalar TensorFlow, este no sobreescribirá versiones preexistentes de paquetes Python requeridos por TensorFlow.

Primero se debe instalar Pip y Virtualenv si no están ya instalados:

```
# Ubuntu/Linux 64-bit
$ sudo apt-get install python-pip python-dev python-virtualenv

# Mac OS X
$ sudo easy_install pip
$ sudo pip install --upgrade virtualenv
```

A continuación se debe crear un entorno virtual Virtualenv. Los siguientes comandos crean un entorno virtual Virtualenv en el directorio ~/tensorflow:

```
$ virtualenv --system-site-packages ~/tensorflow
```

El siguiente paso es activar el Virtualenv de la siguiente manera:

```
$ source ~/tensorflow/bin/activate # si se usa bash
$ source ~/tensorflow/bin/activate.csh # si se usa csh
(tensorflow)$
```

Si nos fijamos, a partir de este momento tenemos delante de la línea de comandos el nombre del entorno virtual en el que nos encontramos trabajando. Una vez activado, se usa Pip para instalar TensorFlow dentro de este.

```
# Ubuntu/Linux 64-bit, CPU only:
(tensorflow)$ pip install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.5.0-cp27-none-linux_x86_6

# Mac OS X, CPU only:
(tensorflow)$ pip install --upgrade https://storage.googleapis.com/tensorflow/mac/tensorflow-0.5.0-py2-none-any.whl
```

Si la plataforma en la que estamos ejecutando nuestro código dispone de GPU, el paquete que debe instalarse para poder usarla es distinto**.** Le recomiendo visitar la documentación oficial[10] para ver si su GPU cumple con las especificaciones necesarias para soportar Tensorflow. Es necesaria la instalación de software adicional para ejecutar Tensorflow en la GPU.

Para más detalle sobre el uso de las GPUs, sugiero al lector que lea el capítulo correspondiente en este libro. Por su carácter introductorio, dejo este tema para más adelante y solo para aquellos lectores que quieran profundizar en ello.

Aunque simplemente con este comando la instalación debería de ser correcta, se recomienda al lector que visite la página de la documentación oficial para asegurarse de que está instalando la última versión disponible. Dado que el código TensorFlow lleva pocas semanas liberado en el momento de escribir este libro, es muy probable que vayan saliendo nuevas versiones.

Finalmente, cuando se ha terminado, se debe desactivar el entorno virtual con:

```
(tensorflow)$ deactivate
```

Dado el carácter introductorio de este libro sugiero al lector visitar la página *Download and Setup TensorFlow*[11] para encontrar más información sobre otras maneras de instalar Tensorflow.

Primer código en TensorFlow

Como ya hemos comentado al principio, vamos a avanzar en esta exploración del planeta TensorFlow con poca teoría y mucha práctica, por lo cual estoy siempre teniendo en mente cuando escribo que el lector está leyendo este libro con un teclado delante.

A partir de este momento, lo mejor será usar cualquier editor de texto para escribir el código de demostración y guardarlo con extensión ".py" (por ejemplo *test.py*). Para ejecutarlo bastará con el comando *python test.py*.

Para hacerse una primera idea de cómo es un código TensorFlow les propongo programar una simple multiplicación; el código es el siguiente:

```
import tensorflow as tf

a = tf.placeholder("float")
b = tf.placeholder("float")

y = tf.mul(a, b)

sess = tf.Session()

print sess.run(y, feed_dict={a: 3, b: 3})
```

En este código, después de importar el módulo tensorflow en Python, definamos variables "simbólicas", llamadas *placeholder* en TensorFlow, para poder manipularlas durante la ejecución del programa. A continuación, pasamos estas variables como parámetro en la llamada a la función de multiplicar que nos ofrece TensorFlow. *tf. mul* es una de las muchas operaciones matemáticas que TensorFlow ofrece sobre sus datos (tensors). Las principales son:

Operación matemática	Descripción
tf.add	Calcula la suma
tf.sub	Calcula la resta
tf.mul	Calcula la multiplicación
tf.div	Calcula la división
tf.mod	Retorna el módulo
tf.abs	Retorna el valor absoluto
tf.neg	Retorna el valor negativo
tf.sign	Retorna el signo
tf.inv	Retorna el inverso
tf.square	Calcula el cuadrado
tf.round	Retorna el valor entero más próximo
tf.sqrt	Calcula la raíz cuadrada
tf.pow	Calcula la potencia
tf.exp	Calcula el exponencial
tf.log	Calcula el logaritmo
tf.maximum	Retorna el máximo
tf.minimum	Retorna el mínimo
tf.cos	Calcula el coseno

tf.sin	Calcula el seno

TensorFlow también pone a disposición del programador varias funciones para realizar operaciones matemáticas sobre matrices. Algunas de ellas son:

Operación	Descripción
tf.diag	Retorna un *tensor* diagonal con unos valores pasados como argumentos
tf.transpose	Retorna el *tensor* pasado como argumento traspuesto
tf.matmul	Retorna un *tensor* resultado de multiplicar dos *tensores* indicados como argumentos
tf.matrix_determinant	Retorna el determinante de la matriz cuadrada indicada como argumento
tf.matrix_inverse	Retorna la inversa de la matriz cuadrada indicada como argumento

El siguiente paso, uno de los más importantes, consiste en crear una sesión para evaluar la expresión simbólica especificada. En realidad, hasta este momento no se ha ejecutado nada de lo especificado en este código TensorFlow. Permítanme insistir en que TensorFlow es a la vez una interficie para expresar algoritmos *Machine Learning* como una implementación para ejecutarlos, y este es un buen ejemplo.

Los programas interactúan con TensorFlow creando una *Session()*; es solo a partir de que la creación de esta que podemos llamar al método *run*, y es cuando realmente empieza a ejecutarse el código especificado. En este ejemplo en concreto, los valores de las variables se introducen en el momento de hacer la llamada al método *run* con el argumento *feed_dict*. Es entonces cuando se ejecuta el código asociado a resolver la expresión y sale por la pantalla el valor 9 resultado de la multiplicación.

Con este simple ejemplo se ha querido introducir la idea de que la forma habitual de programar en TensorFlow es especificar primero todo el problema, y al final crear la sesión para permitir ejecutar la computación asociada mediante llamadas a sus métodos asociados.

Pero, a veces, nos interesa más flexibilidad a la hora de estructurar el código, intercalando operaciones para construir el grafo con operaciones que ejecutan parte de él. Ocurre cuando estamos, por ejemplo, usando entornos interactivos de Python como IPython. Para ello TensorFlow ofrece la clase tf.*InteractiveSession ()*.

Más adelante entraremos en la motivación por este modelo de programación. De momento, para poder seguir con el siguiente capítulo, solo debemos saber que toda la información TensorFlow se guarda internamente en una estructura en grafo que contiene toda la información de las operaciones y los datos.

Con el flujo de datos, este grafo describe las computaciones matemáticas con un grafo de nodos y aristas. Los nodos típicamente implementan operaciones matemáticas, pero también pueden representar puntos de entrada de datos, salida de resultados, o lecturas/escritura de variables persistentes. Las aristas describen las relaciones entre nodos con sus entradas y salidas y transportan los tensores, la estructura de datos básica de TensorFlow.

La representación de la información en forma de grafo facilita conocer las dependencias entre operaciones, y permite asignar operaciones a los dispositivos de manera asíncrona y en paralelo cuando estas operaciones tienen los *tensores* asociados a todas sus aristas de entrada ya disponibles.

El paralelismo es, pues, uno de los factores por los que se consigue ejecutar de forma muy rápida algunos algoritmos costosos computacionalmente, pero también porque tiene un conjunto de operaciones ya implementadas de forma eficiente que representan operaciones complejas del estilo de multiplicación de matrices. Pero además, la mayoría de estas operaciones tienen *kernels* asociados, que son implementaciones de operaciones diseñadas para dispositivos específicos

como pueden ser GPUs. En la siguiente tabla se presenta un resumen de las operaciones/*kernels* más importantes[12]:

Categoría	Operaciones principales
Operaciones matemáticas básicas	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal
Operaciones en Array	Concat, Slice, Split, Constant, Rank, Shape, Shuffle
Operaciones de matrices	MatMul, MatrixInverse, MatrixDeterminant
Operaciones para redes neuronales	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool
Operaciones de Checkpointing	Save, Restore
Operaciones de colas y sincronizaciones	Enqueue, Dequeue, MutexAcquire, MutexRelease
Operaciones de control de flujo	Merge, Switch, Enter, Leave, NextIteration

Panel de visualización Tensorboard

Para que sea más fácil de entender, TensorFlow ha incluido las funciones de depurar y optimizar los programas en una herramienta de visualización llamada TensorBoard. TensorBoard permite visualizar diferentes tipos de estadísticas sobre los parámetros, así como detalles de cualquier parte del grafo de computación de una manera gráfica.

Los datos que se visualizan con TensorBoard se generan durante la ejecución de TensorFlow y se guardan en ficheros de trazas cuyos datos se obtienen de las *summary operations*. En la página de documentación de TensorFlow pueden encontrar la explicación detallada de la API en Python[13] para capturar la información que se requiera.

La manera de usarlo es muy simple: se crea un servicio con los comandos Tensorflow desde la línea de comandos, en el que se le incluye como argumento el fichero de la traza, y simplemente hace falta acceder al *socket* local 6006 desde el navegador con http://localhost:6006/ (recomiendo usar Google Chrome para garantizar una correcta visualización).

Para más detalles sobre el funcionamiento, el lector puede visitar el apartado *TensorBoard Graph Visualization*[14] de la página del tutorial de TensorFlow.

[volver al índice de contenidos]

2. Regresión lineal

En este capítulo, empezaremos a explorar TensorFlow con uno de los modelos más sencillos y seguramente uno de los familiares al lector: la regresión lineal. Como veremos a continuación en un problema de regresión, para cada uno de los valores de entrada se obtiene el valor de salida en un espacio continuo, en un contexto de aprendizaje supervisado, puesto que conocemos el tipo de valores de salida esperados.

El objetivo de este capítulo es permitir al lector hacer una primera aproximación intuitiva a dos piezas fundamentales en *Machine Learning*, la función de coste y el algoritmo *gradient descent* usando como ejemplo práctico una regresión lineal.

Modelo de relación entre variables

La regresión lineal es una técnica estadística utilizada para medir la relación entre variables. Su interés radica en que el algoritmo que lo implementa no es complejo conceptualmente, y además se adapta a una amplia variedad de situaciones. Por esos motivos me ha parecido interesante empezar a profundizar con TensorFlow con un ejemplo de regresión lineal.

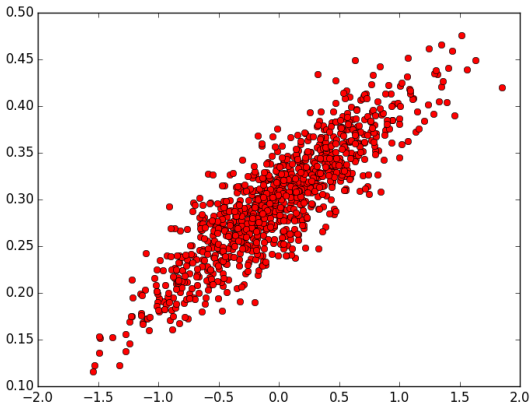
Además, dada la popularidad del algoritmo, entiendo que al lector del libro le es familiar este algoritmo. Recordemos que tanto en el caso de dos variables (regresión simple) como en el de más de dos variables (regresión múltiple), la regresión lineal modela la relación entre una variable dependiente *y*, las variables independientes x_i y un término aleatorio *b*.

En este apartado crearemos un ejemplo sencillo para explorar cómo funciona TensorFlow suponiendo que nuestro modelo de datos corresponde a una regresión lineal simple: *y = W * x + b*. Para ello, usaremos un pequeño programa en Python que crea datos en un espacio de dos dimensiones y, a continuación, pediremos a TensorFlow que busque la recta que mejor ajuste a estos puntos.

Lo primero será importar el paquete NumPy que se usará para la generación de puntos. El código que hemos creado es el siguiente:

```
import numpy as np

num_puntos = 1000
conjunto_puntos = []
for i in xrange(num_puntos):
        x1= np.random.normal(0.0, 0.55)
        y1= x1 * 0.1 + 0.3 + np.random.normal(0.0, 0.03)
        conjunto_puntos.append([x1, y1])

x_data = [v[0] for v in conjunto_puntos]
y_data = [v[1] for v in conjunto_puntos]
```

Como se desprende del código, se han generado unos puntos que siguen la relación *y = 0.1 * x + 0.3*, eso sí, con una cierta variación, usando una distribución normal para que los puntos no se ajusten exactamente a una recta y nos permita hacer un ejemplo más interesante. En nuestro caso, una visualización de la nube de puntos resultante es la siguiente:



Si quiere, el lector puede visualizarlos con el siguiente código (en este caso, hace falta importar algunas de las funciones del paquete *matplotlib*; ejecutando *pip install matplotlib* instalaremos previamente el paquete[15]):

```
import matplotlib.pyplot as plt


plt.plot(x_data, y_data, 'ro', label='Original data')
plt.legend()
plt.show()
```

Estos datos son los que consideraremos datos de entrenamiento para nuestro modelo.

Función de coste y algoritmo *gradient descent*

El siguiente paso consiste en entrenar nuestro algoritmo de aprendizaje para que sea capaz de precedir los valores de salida *y,* estimados a partir de la entrada *x_data*. En este caso, como sabemos de antemano que se trata de una regresión lineal, podemos representar nuestro modelo con solo dos parámetros *W* y *b*.

Se trata de generar un código en TensorFlow que permita encontrar los mejores parámetros *W* y *b*, que a partir de los datos de entrada *x_data*, ajuste de la mejor manera a los datos de salida *y_data*, en nuestro caso en forma de una recta definida por *y_data = W * x_data + b* . El lector sabe que el valor *W* debe ser próximo al 0.1 y *b* próximo a 0.3, pero TensorFlow no lo sabe y debe darse cuenta de ello por si solo.

Una forma estándar de solucionar este tipo de problemas es iterar a través de cada valor del conjunto de datos e ir modificando los parámetros *W* y *b*, para obtener una respuesta cada vez más acertada. Para saber si vamos mejorando en estas iteraciones, definiremos una función de coste (también llamada función de error) que mida como de "buena" (en realidad como de "mala") es una determinada recta. Esta función recibe como parámetros el par *W y b,* y devuelve un valor de error basado en cómo de bien la recta ajusta a los datos. En nuestro ejemplo podemos usar como función de coste la que se conoce como *mean squared error*[16].

Con el *mean squared error* obtenemos la media de los "errores" en base a la distancia entre el valor real y el valor estimado de cada una de la iteraciones. Más adelante entraremos en detalle con la función de coste y sus alternativas, pero para este ejemplo introductorio nos sirve para ir avanzando paso a paso.

Vamos ahora a programar en TensorFlow lo contado hasta el momento. Para ello primero creamos tres variables con la siguientes sentencias:

```
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b
```

De momento, podemos avanzar sabiendo solo que con la llamada al método *Variable* estamos definiendo una variable que reside en el grafo de operaciones que construye TensorFlow internamente del que hemos hablado anteriormente. Como parámetros, reciben los valores con los que inicializaremos estas variables. Más adelante volveremos a este tema y entraremos con más detalle, pero de momento creo que es mejor avanzar para facilitar esta primera aproximación.

Ahora, con estas variables definidas, ya podemos expresar la función de coste que hemos comentado anteriormente, basada en la distancia que hay entre cada punto y el valor candidato calculado a partir de la función y= *W * x + b*. Después se calcula su cuadrado y se hace la media. En TensorFlow esta función de coste se expresa de la siguiente manera:

```
loss = tf.reduce_mean(tf.square(y - y_data))
```

Como vemos, esta expresión calcula la media de las distancias al cuadrado del punto de salida *y_data* que conocemos y el punto *y* calculado a partir del *x_data* de entrada.

En este momento, el lector ya intuye que la recta que mejor se ajusta a nuestros datos es la que consigue un valor de error menor. Por tanto, si minimizamos esta función de error, encontraremos el mejor modelo para nuestros datos.

Sin de momento entrar en mucho detalle, diremos que esto es lo que el algoritmo de optimización conocido por *gradient descent*[17] consigue. Empieza con unos valores iniciales de un conjunto de parámetros (en nuestro caso *W* y *b*), que iterativamente va modificando el valor para cada uno de estos parámetros de tal manera que acaben teniendo unos valores que minimicen la función de coste.

Más adelante ya entraremos en más detalle, pero de momento vamos a usar nuestro ejemplo de regresión lineal para visualizar su funcionamiento. Para usar este algoritmo en TensorFlow solo tenemos que ejecutar las siguientes dos instrucciones:

```
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)
```

Nos podemos quedar con la idea que TensorFlow ha creado la información pertinente en su estructura de datos interna, que implementa un optimizador que podrá ser referenciado por *train*, un algoritmo *gradient descent* con la función de coste definida. De momento no entramos en detalle sobre el parámetro de la función (conocida como *learning rate*, de la que más adelante hablaremos).

Ejecución del algoritmo

Como ya hemos visto, hasta este punto en el código las llamadas a la librería TensorFlow van añadiendo información al grafo interno pero todavía no se han ejecutado ninguno de los algoritmos. Por ello, al igual que en el anterior ejemplo, debemos crear una sesión para poder llamar a *run* pasándole como parámetro *train*. En este ejemplo, además, puesto que en el código hemos especificado variables, debemos inicializarlas previamente con las siguientes llamadas:

```
init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)
```

Ya podemos empezar con el proceso iterativo que nos permitirá encontrar los valores de *W* y *b* que definen el modelo de recta que mejor se ajusta a los puntos de entrada. En nuestro ejemplo en concreto, supongamos que considerando que con unas 8 iteraciones es suficiente, el código podría ser el siguiente:

```
for step in xrange(8):
    sess.run(train)
print step, sess.run(W), sess.run(b)
```

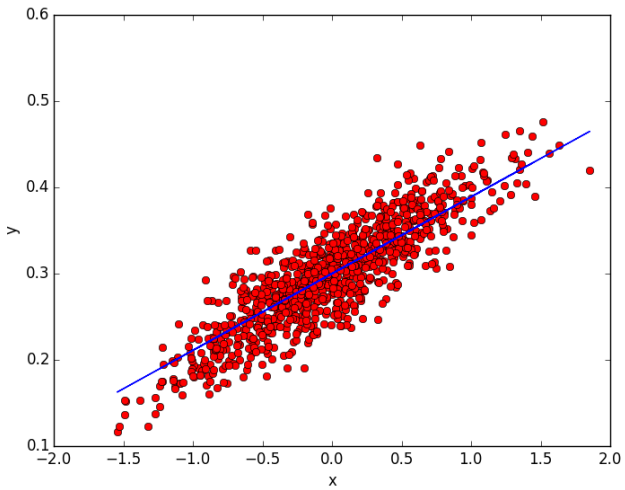
Si lo ejecutan, les deberían salir unos resultados en que los valores de *W* y *b* se acercan al valor que nosotros conocemos de antemano. En mi caso, el resultado del *print* es:

```
(array([ 0.09150752], dtype=float32), array([ 0.30007562], dtype=float32))
```

Y, si volvemos a visualizar gráficamente el resultado con el código:

```
plt.plot(x_data, y_data, 'ro')
plt.plot(x_data, sess.run(W) * x_data + sess.run(b))
plt.legend()
plt.show()
```

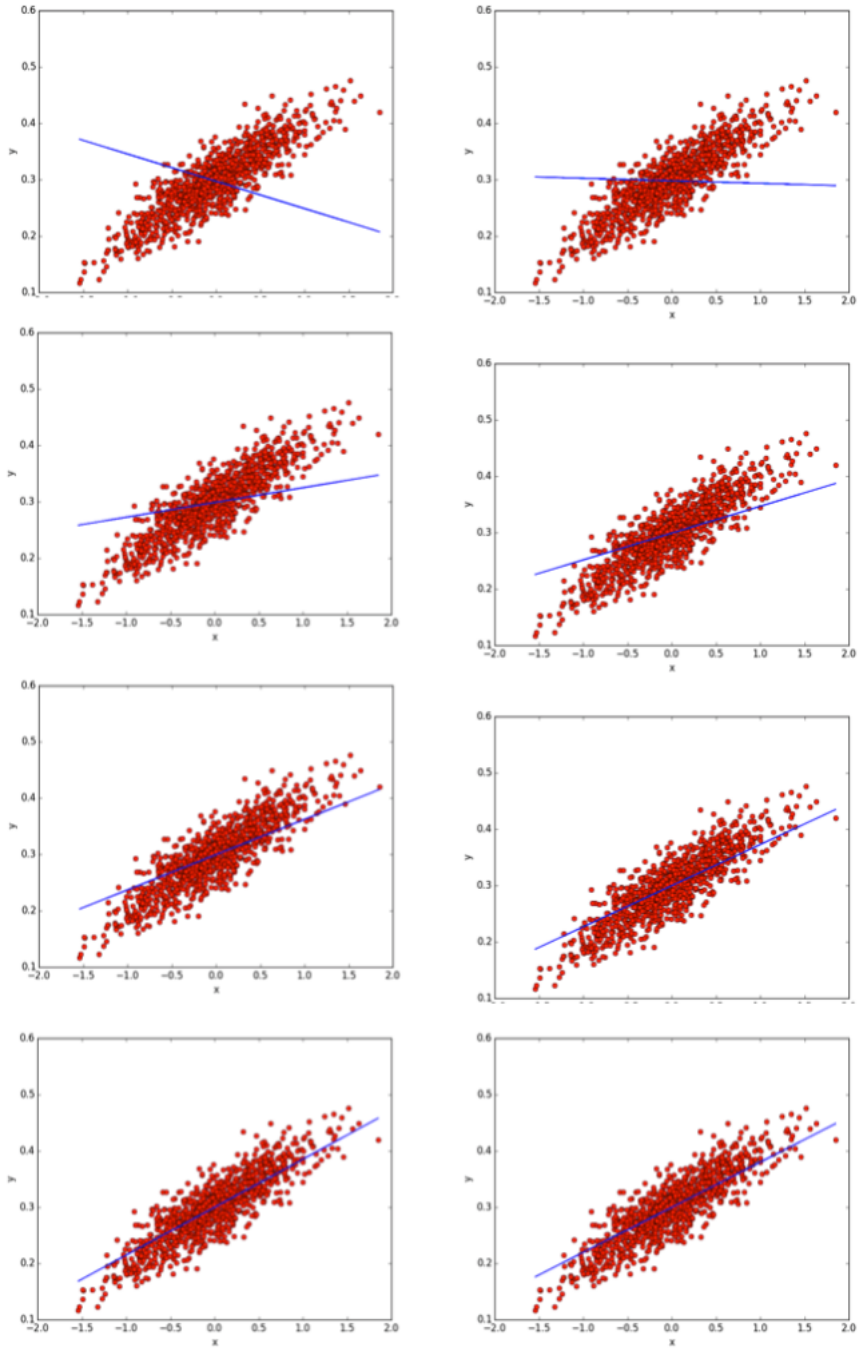
Podemos ver gráficamente la recta definida por los parámetros W=0.0915 y b= 0.3 encontrados:



Remarcar que solo hemos ejecutado 8 iteraciones para facilitar la explicación, pero si ejecutamos más, el algoritmo conseguiría aproximar más los valores de los parámetros a los esperados.

Le propongo al lector visualizar esta gráfica en cada iteración, que permite ver visualmente cómo el algoritmo va afinando los valores de los parámetros. En nuestro caso las gráficas y la salida del *print* son:

```
(0, array([-0.04841119], dtype=float32), array([ 0.29720169], dtype=float32))
(1, array([-0.00449257], dtype=float32), array([ 0.29804006], dtype=float32))
(2, array([ 0.02618564], dtype=float32), array([ 0.29869056], dtype=float32))
(3, array([ 0.04761609], dtype=float32), array([ 0.29914495], dtype=float32))
(4, array([ 0.06258646], dtype=float32), array([ 0.29946238], dtype=float32))
(5, array([ 0.07304412], dtype=float32), array([ 0.29968411], dtype=float32))
(6, array([ 0.08034936], dtype=float32), array([ 0.29983902], dtype=float32))
(7, array([ 0.08545248], dtype=float32), array([ 0.29994723], dtype=float32))
```



Como puede verse, a cada iteración el algoritmo ajusta más la recta a los datos. Más adelante veremos con más detalle cómo el algoritmo *gradient descent* consigue ir acercándose a los valores de los parámetros que minimizan la función de coste.

Recuerden que para facilitar al lector el probar el código descrito en este capítulo lo puede descargar del *github* del libro[18] con el nombre de *regresion.py*. A continuación lo encontrarán todo junto para facilitar su seguimiento:

```
import numpy as np

num_puntos = 1000
conjunto_puntos = []
```

```
for i in xrange(num_puntos):
     x1= np.random.normal(0.0, 0.55)
     y1= x1 * 0.1 + 0.3 + np.random.normal(0.0, 0.03)
     conjunto_puntos.append([x1, y1])

x_data = [v[0] for v in conjunto_puntos]
y_data = [v[1] for v in conjunto_puntos]

import matplotlib.pyplot as plt

#Graphic display
plt.plot(x_data, y_data, 'ro')
plt.legend()
plt.show()

import tensorflow as tf

W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)

for step in xrange(16):
        sess.run(train)
        print(step, sess.run(W), sess.run(b))
        #Graphic display
        plt.plot(x_data, y_data, 'ro')
        plt.plot(x_data, sess.run(W) * x_data + sess.run(b))
        plt.xlabel('x')
        plt.xlim(-2,2)
        plt.ylim(0.1,0.6)
        plt.ylabel('y')
        plt.legend()
        plt.show()
```

En este capítulo hemos empezado a explorar las posibilidades de TensorFlow con una primera aproximación intuitiva a dos piezas fundamentales en *Machine Learning*: la función de coste y el algoritmo *gradient descent*, usando como marco de explicación el algoritmo de regresión lineal. En el siguiente capítulo profundizaremos y entraremos en más detalle a las estructuras de datos que usa TensorFlow.

[volver al índice de contenidos]

3. Clustering con K-means

La regresión lineal presentada en el capítulo anterior es un algoritmo de aprendizaje supervisado en que utilizamos la muestra de datos y sus valores de salida (o etiquetas) para construir un modelo que se adapte a ellos. Pero no siempre tenemos los datos etiquetados y, a pesar de ello, queremos analizarlos de alguna manera. En este caso, se puede usar un algoritmo de aprendizaje no supervisado: la agrupación (o *clustering*). El *clustering* es un método muy utilizado porque a menudo se utiliza para el análisis exploratorio previo de los datos.

En este capítulo voy a presentar el algoritmo de *clustering* llamado *K-means*, seguramente el más popular y ampliamente utilizado para agrupar automáticamente los datos en subconjuntos

coherentes de tal manera que todos los elementos en un subconjunto sean más similares entre ellos que con el resto.

Aprovecharé este capítulo para avanzar también en el conocimiento de TensorFlow, y entraremos más detalladamente en la estructura de datos básica *tensor*. Empezaré explicando cómo es este tipo de datos y las transformaciones que podemos realizar sobre él, y después mostraré el uso del algoritmo *K-means* en un caso concreto del uso de los tensores.

Estructura de datos básica: *tensor*

Los programas TensorFlow usan una estructura de datos básica llamada *tensor* para representar todos sus datos. Un *tensor* se puede ver como un *array* o lista n-dimensional que tiene como propiedades un tipo (*type*) estático de datos, que puede ser desde *booleano* o *string* hasta una gran variedad de tipos numéricos. A continuación, se presenta una tabla con los principales tipos y su equivalencia en Python.

Tipo en TensorFlow	Tipo en Python	Descripción
DT_FLOAT	tf.float32	Coma flotante de 32 bits
DT_INT16	tf.int16	Entero de 16 bits
DT_INT32	tf.int32	Entero de 32 bits
DT_INT64	tf.int64	Entero de 64 bits
DT_STRING	tf.string	*String*
DT_BOOL	tf.bool	*Booleano*

Además, cada *tensor* tiene un rango (*rank*), que es el número de dimensiones del *tensor*. Por ejemplo, el siguiente *tensor* (definido como una lista en Python) tiene *rank* 2:

```
t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Los *tensors* pueden tener cualquier *rank*. Un *tensor* de *rank* 2 es lo que habitualmente consideramos una matriz, y un *tensor* de *rank* 1 sería un vector. Un valor escalar se considera de *rank* 0.

La documentación de TensorFlow usa tres tipos de convenciones notacionales para describir la dimensionalidad de un *tensor*: *rank*, forma (*shape)* y número de dimensión (*Dimension Number*). En la siguiente tabla se indica la relación entre ellos para facilitar el seguimiento de la documentación de TensorFlow:

Shape	*Rank*	*Dimension Number*
[]	0	0-D
[D0]	1	1-D
[D0, D1]	2	2-D
[D0, D1, D2]	3	3-D
…	…	…
[D0, D1, … Dn]	n	n-D

Estos *tensores* pueden manipularse con una serie de transformaciones que ofrece el paquete TensorFlow. A continuación, comentamos algunas de ellas en la tabla adjunta, creo que suficientes dado el carácter introductorio del libro.

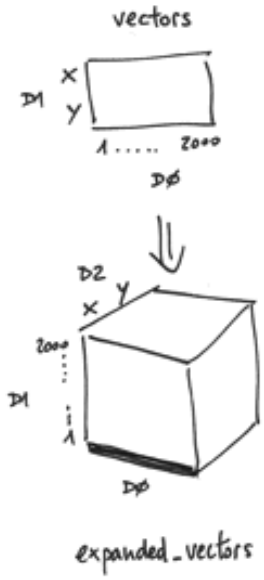
A lo lago del capítulo entraremos en más detalle en algunas de ellas. La lista completa de transformaciones, así como más detalles de cada una, puede encontrarse en la página oficial de TensorFlow, *Tensor Transformations*[19].

Operación	Descripción
tf.shape	Para saber la forma de un *tensor*
tf.size	Para saber el número de elementos de un *tensor*
tf.rank	Para saber el rango de un *tensor*
tf.reshape	Permite cambiar la forma de un *tensor* manteniendo los mismos elementos que contenía
tf.squeeze	Permite borrar del *tensor* dimensiones de tamaño 1
tf.expand_dims	Permite insertar una dimensión al *tensor*
tf.slice	Extrae una porción del *tensor*
tf.split	Divide el *tensor* en varios *tensores* a lo largo de una dimensión
tf.tile	Crea un nuevo *tensor* replicando un *tensor* múltiples veces
tf.concat	Concatena *tensores* en una de las dimensiones
tf.reverse	Invierte una determinada dimensión de un *tensor*
tf.transpose	Permuta dimensiones de un *tensor*
tf.gather	Recolecta porciones de acuerdo a un índice

Por ejemplo, supongamos que queremos extender una matriz de 2×2000 (un *tensor* 2D) a un cubo (*tensor* 3D). Podemos usar la función *tf.expand_dims*, que permite insertar una dimensión a un *tensor*.

```
vectors = tf.constant(conjunto_puntos)
extended_vectors = tf.expand_dims(vectors, 0)
```

En este caso, *tf.expand_dims* inserta una dimensión en el *tensor* en la dimensión indicada en el argumento (las dimensiones empiezan en zero). Visualmente, la anterior transformación es la siguiente:



Como vemos, ahora tenemos un *tensor* 3D, pero de los argumentos pasados a la función no se puede determinar el tamaño de la nueva dimensión D0.

Obteniendo la forma de este *tensor* con la operación *.get_shape()* podemos comprobar que no hay tamaño asociado:

```
print expanded_vectors.get_shape()
```

Que nos aparece por pantalla:

```
TensorShape([Dimension(1), Dimension(2000), Dimension(2)])
```

Más adelante, en este mismo capítulo veremos que, gracias a que TensorFlow permite hacer *shape broadcasting,* muchas funciones matemáticas de manipulación de *tensores* como las presentadas en el primer capítulo son capaces de descubrir por sí mismas el tamaño en la dimensión que nos falta su especificación y asignarle el valor correspondiente.

Almacenamiento de datos en TensorFlow

Siguiendo con la presentación del paquete TensorFlow, en lo que se refiere a datos podemos resumir que hay tres formas principales para la obtención de datos en un programa TensorFlow:

1. Desde ficheros de datos al comienzo de un programa TensorFlow.
2. Datos precargados en forma de constantes o variables.
3. Los que proporciona el código Python.

A continuación, vamos a describir cada uno de ellos para que el lector disponga de una visión global de las opciones de que dispone para alimentar de datos sus algoritmos.

1. **Ficheros de datos**

Habitualmente, los datos iniciales se descargan de un fichero de datos. El proceso no es complejo, y dado el carácter introductorio de este libro, invito al lector a visitar la página web de TensorFlow[20] para más detalles sobre cómo descargar los datos de diferentes tipos de ficheros. También puede repasar el código Python *input_data.py* ***[21]*** (disponible en el *github* del libro), que carga desde ficheros los datos MNIST que usaré en los siguientes capítulos.

2. **Variables y constantes**

Cuando se trata de conjuntos pequeños, los datos también se pueden encontrar precargados en memoria; hay dos maneras básicas de crearlos, como ya hemos visto en algún ejemplo anterior:

- como constantes con la función *constant(…)*
- como variables con la función *Variable(…)*

El paquete de TensorFlow ofrece diferentes operaciones que se pueden usar para generar constantes. En la siguiente tabla se puede encontrar un resumen de las más importantes:

Operación	Descripción
tf.zeros_like	Crea un *tensor* con todos los elementos inicializados a 0
tf.ones	Crea un *tensor* con todos los elementos inicializados a 1
tf.ones_like	Crea un *tensor* con todos los elementos inicializados a 1
tf.fill	Crea un *tensor* con todos los elementos inicializados a un valor *scalar* pasado como argumento
tf.constant	Crea un *tensor* de constantes con los elementos indicados como argumento

En TensorFlow, para entrenar a los modelos se acostumbra a mantener los parámetros como *tensores* en memoria en forma de variables. Cuando se crea una variable, se le pasa un *tensor* argumento del constructor como valor inicial, que puede ser un *tensor* tipo *constant* o *tensores* con valores *random*. TensorFlow ofrece una colección de operaciones que producen *tensores random* con diferentes distribuciones:

Operación	Descripción
tf.random_normal	Valores *random* con una distribución normal, con una media y desviación estándar indicadas como argumentos
tf.truncated_normal	Igual que la anterior pero eliminando aquellos valores cuya magnitud es más de 2 veces la desviación estándar.
tf.random_uniform	Valores *random* con una distribución uniforme de un rango indicado en los argumentos
tf.random_shuffle	Mezclados aleatoriamente los elementos de un *tensor* en su primera dimensión
tf.set_random_seed	Establece la semilla aleatoria a nivel de grafo

Un detalle importante es que todas estas operaciones requieren especificar la forma (*shape*) del *tensor* en los argumentos, y esta misma forma es la que tendrá la variable que se crea con esta

operación. En general, las variables tienen una forma fija, pero TensorFlow provee mecanismos de remodelar la forma si fuera necesario.

Cuando usamos variables, recordemos que estas se tienen que iniciar explícitamente después de que el grafo haya sido construido y antes de que se ejecute alguna operación con la función *run()*. Como hemos visto, se puede usar *tf.initialize_all_variables()* para este propósito. Las variables, además, pueden ser guardadas en disco durante y después del entrenamiento del modelo, mediante la clase de TensorFlow *tf.train.Saver*, pero esta clase queda fuera del alcance de este libro.

3. **Desde el código Python**

Finalmente, podemos usar lo que hemos llamado previamente "variable simbólica" o *placeholder* para manipular los datos durante la ejecución del programa. La llamada usada es *placeholder()*, con la que se le pasa como argumento el tipo de los elementos y la forma del *tensor*, y de forma opcional un nombre.

Pero no es hasta el momento de hacer las llamadas a *Session.run()* o *Tensor.eval()* desde el código de Python que se rellena este *tensor* con los datos que se especifican en el argumento *feed_dict* de esta función. Recordemos el primer código en TersorFlow de este libro en el capítulo 1:

```
import tensorflow as tf
a = tf.placeholder("float")
b = tf.placeholder("float")
y = tf.mul(a, b)
sess = tf.Session()
print   sess.run(y, feed_dict={a: 3, b: 3})
```

En la última línea del código, cuando se realiza la llamada *sess.run*, se le pasan los valores de los dos *tensores a* y *b* a través del argumento *feed_dict*.

Espero que el lector, con esta sección de introducción a los *tensores* de TensorFlow le permita seguir sin dificultad los códigos de los siguientes capítulos.

Algoritmo *K-means* en TensorFlow

K-means es un método que tiene como objetivo la partición de un conjunto de observaciones en *K* grupos (*clusters*), en el que cada observación (o punto) pertenece al grupo más cercano. El resultado del algoritmo es un conjunto de K puntos, llamados centroides, que son el punto central de los diferentes grupos obtenidos, y un etiquetado del conjunto de puntos que los asigna a solamente uno de los K *clusters*. Se cumple que todos los puntos dentro de un *cluster* están más cerca en distancia a su centroide que a cualquiera de los otros centroides.

Hacer *clusters* es un problema computacionalmente costoso si se quiere minimizar la función de error directamente (lo que se conoce como un problema *NP-hard*), y por ello se han creado algoritmos que convergen rápidamente en un óptimo local mediante heurísticas. El algoritmo más comúnmente utilizado usa una técnica de refinamiento iterativo que converge en pocas iteraciones.

A grandes rasgos esta técnica tiene estos tres pasos:

- **Paso inicial (paso 0)**: determina un conjunto inicial de *K*
- **Paso de asignación (paso 1):** asigna cada observación al grupo más cercano.
- **Paso de actualización (paso 2)**: calcula los nuevos centroides como el centroidede las observaciones que han sido agrupados al grupo.

Hay varios métodos para determinar los K grupos iniciales o K centroides. Uno de ellos es elegir aleatoriamente K observaciones del conjunto de datos y considerarlos centroides; este es el que usaremos en nuestro ejemplo.

Los pasos de asignación y actualización se van alternando en un bucle hasta que se considera que el algoritmo ha convergido, que puede ser por ejemplo cuando las asignaciones de los puntos a grupos ya no cambian.

Como se trata de un algoritmo heurístico, no hay ninguna garantía de que se converja al óptimo global, y el resultado depende de los grupos iniciales. Por ello, como el algoritmo suele ser muy

rápido, es habitual repetir las ejecuciones varias veces con diferentes valores de los centroides iniciales, y luego ponderar el resultado.

Para empezar a programar nuestro ejemplo de *K-means* en TensorFlow generaremos primero algunos datos como juego de pruebas en el propio código Python. Propongo hacer algo simple, y generar 2.000 puntos en un espacio 2D de manera *random*, siguiendo dos distribuciones normales que nos dibujen un espacio que nos permita entender bien el resultado. Por ejemplo, les propongo el siguiente código:

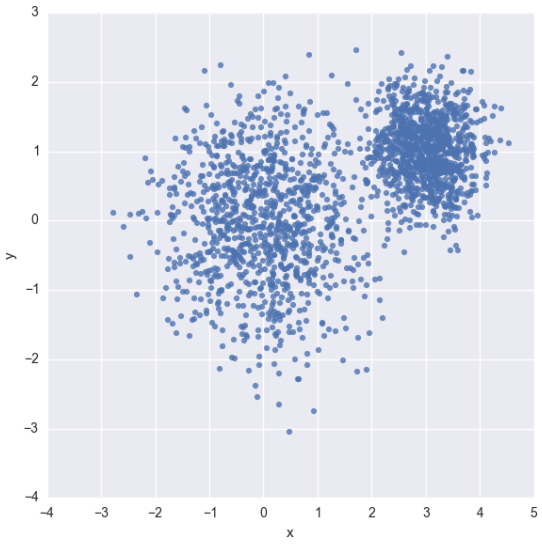
```
num_puntos = 2000
conjunto_puntos = []
for i in xrange(num_puntos):
   if np.random.random() &gt; 0.5:
     conjunto_puntos.append([np.random.normal(0.0, 0.9), np.random.normal(0.0, 0.9)])
   else:
     conjunto_puntos.append([np.random.normal(3.0, 0.5), np.random.normal(1.0, 0.5)])
```

Igual que hemos hecho en el anterior capítulo, miremos gráficamente nuestro juego de pruebas usando algunas librerías gráficas de Python. Les propongo que usemos, igual que antes, la popular librería *matplotlib*, aunque esta vez también utilizaremos el paquete de visualización *Seaborn* basado en *matplotlib* y el paquete de manipulación de datos *pandas*, que permite trabajar con estructuras de datos más complejas. Si no tiene estos paquetes instalados, deberá hacerlo con la utilidad *pip* antes de poder ejecutar los siguientes códigos.

Para visualizar los puntos que se han generado aleatoriamente les propongo el siguiente código:

```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

df = pd.DataFrame({"x": [v[0] for v in conjunto_puntos],
        "y": [v[1] for v in conjunto_puntos]})
sns.lmplot("x", "y", data=df, fit_reg=False, size=6)
plt.show()
```

Que crea un gráfico de los puntos en un espacio de dos dimensiones como el siguiente:



Un algoritmo *k-means* implementado en TensorFlow para agrupar los anteriores puntos en, por ejemplo, 4 *clusters*, puede ser el siguiente (basado en el propuesto por Shawn Simister en su blog[22]):

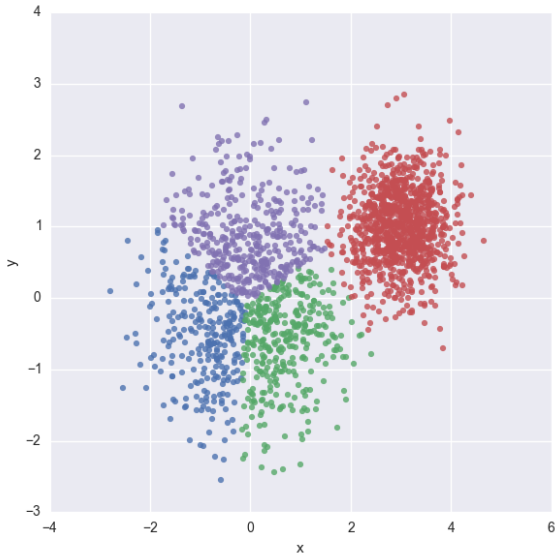
```
import numpy as np

vectors = tf.constant(conjunto_puntos)
k = 4
centroides = tf.Variable(tf.slice(tf.random_shuffle(vectors),[0,0],[k,-1]))
```

```
expanded_vectors = tf.expand_dims(vectors, 0)
expanded_centroides = tf.expand_dims(centroides, 1)

assignments = tf.argmin(tf.reduce_sum(tf.square(tf.sub(expanded_vectors, expanded_centroides)), 2), 0)

means = tf.concat(0, [tf.reduce_mean(tf.gather(vectors, tf.reshape(tf.where( tf.equal(assignments, c)),[1,-1])), reduction_ind

update_centroides = tf.assign(centroides, means)

init_op = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init_op)

for step in xrange(100):
   _, centroid_values, assignment_values = sess.run([update_centroides, centroides, assignments])
```

Le propongo al lector que compruebe el resultado en el *tensor assignment_values* con el siguiente código, que genera un gráfico como el anterior:

```
data = {"x": [], "y": [], "cluster": []}

for i in xrange(len(assignment_values)):
  data["x"].append(conjunto_puntos[i][0])
  data["y"].append(conjunto_puntos[i][1])
  data["cluster"].append(assignment_values[i])

df = pd.DataFrame(data)
sns.lmplot("x", "y", data=df, fit_reg=False, size=6, hue="cluster", legend=False)
plt.show()
```

La captura de pantalla con el resultado de la ejecución de mi código se muestra en la siguiente figura:



Cálculo de los nuevos grupos

Supongo que el lector se sentirá un poco abrumado con el código *K-means* presentado en la sección anterior. Bien, les propongo que lo analicemos en detalle paso a paso, y en especial veamos como son los *tensores* que intervienen y como estos se transforman a lo largo del programa.

Lo primero que tenemos que hacer es pasar todos nuestros datos a estructuras de datos TensorFlow, es decir, tensores. En un *tensor constant* guardamos nuestros puntos de entrada generados aleatoriamente:

```
vectors = tf.constant(conjunto_vectors)
```

Siguiendo el algoritmo presentado en la sección anterior, para empezar debemos determinar los centroides iniciales. Como ya he avanzado, una opción puede ser elegir aleatoriamente *K* observaciones del conjunto de datos de entrada. Una forma de hacerlo es con el siguiente código, que indica a TensorFlow que debe mezclar aleatoriamente los puntos de entrada y quedarse con los *K* puntos primeros como centroides:

```
k = 4
centroides = tf.Variable(tf.slice(tf.random_shuffle(vectors),[0,0],[k,-1]))
```

Estos *K* puntos se guardan en un *tensor* 2D. Para ver como son los *tensores* recordemos que podemos usar la operación *tf.Tensor.get_shape()* para saber su forma:

```
print vectors.get_shape()
print centroides.get_shape()

TensorShape([Dimension(2000), Dimension(2)])
TensorShape([Dimension(4), Dimension(2)])
```

Se observa que *vectors* es una matriz en que la dimensión D0 contiene 2000 posiciones, una para cada punto, y en D1 la posición *x, y* de los puntos. En cambio, *centroides* es una matriz de 4 posiciones en la dimensión D0, una para cada centroide, y la dimensión D1 es equivalente a la D1 de *vectors*.

Seguidamente, el algoritmo entra en un bucle donde el primer paso, el paso de asignación, consiste en calcular primero por cada punto su centroide más cercano mediante la *Squared Euclidean Distance*[23] (que se puede usar cuando solo queremos comparar distancias):

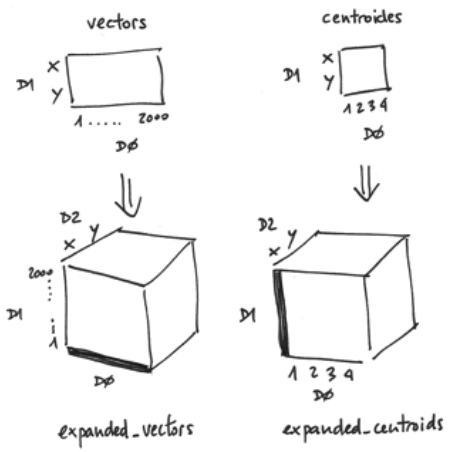
$$d^2(vector, centroide) = (vector_x - centroide_x)^2 + (vector_y - centroide_y)^2$$

Para calcular este valor se utiliza la función *tf.sub(vectors, centroides)* en TensorFlow. Pero si la usamos sin hacer un paso previo, aparece el problema de que los tensores, a pesar de ser los dos de 2 dimensiones, tienen diferentes tamaños en una de las dimensiones (2000 vs 4 en D0), que, en realidad, además representan cosas diferentes, y esto impide poder realizar correctamente la operación resta.

Para solucionarlo se hace uso de algunas de las funciones que hemos comentado anteriormente, en concreto la de *tf.expand_dims* para insertar una dimensión. El objetivo es extender los dos *tensores* 2D a 3D mediante las siguientes instrucciones para poder realizar una sustracción de manera que los tamaños coincidan. Concretamente, se realizan estas dos instrucciones:

```
expanded_vectors = tf.expand_dims(vectors, 0)
expanded_centroides = tf.expand_dims(centroides, 1)
```

tf.expand_dims inserta una dimensión en cada *tensor*, en la primera dimensión (D0) para *vectors* y la segunda dimensión (D1) para *centroides*. Gráficamente, podemos ver que en los *tensores* extendidos las dimensiones tienen el mismo significado:



Parece que esté resuelto, pero en realidad, si nos fijamos (trazado en grueso en la ilustración) hay dimensiones en cada caso, en las que de momento no se han podido determinar los tamaños. Recordemos que con la función *get_shape()* podemos averiguarlo:

```
print expanded_vectors.get_shape()
print expanded_centroides.get_shape()
```

Y que nos aparece por pantalla con los valores respectivamente de:

```
TensorShape([Dimension(1), Dimension(2000), Dimension(2)])
TensorShape([Dimension(4), Dimension(1), Dimension(2)])
```

Indicando con 1 que no tiene tamaño asignado.

Pero ya hemos avanzado que TensorFlow permite hacer *shape broadcasting,* y por ello la función *tf.sub* es capaz de descubrir por sí misma la manera de hacer la sustracción de elementos entre los dos tensores. De manera intuitiva y observando los dibujos anteriores, vemos que la forma de los dos *tensores* coinciden, y en aquellos casos que en ambos *tensores* en una misma dimensión tienen tamaño definido, este coincide (es el caso de la dimensión D2). En cambio, en la dimensión D0 solo tiene tamaño definido el *tensor expanded_centroides*. En este caso, TensorFlow asume que la dimensión D0 de *expanded_vectors* debe tener el mismo tamaño si lo que queremos es realizar una resta elemento a elemento dentro de una dimensión. Y lo mismo ocurre con el tamaño de la dimensión D1 del *tensor expended_centroides*, donde TensorFlow deduce el tamaño de la dimensión D1 del *tensor expanded_vectors*.

Por lo tanto, el paso de asignación (paso 1) del algoritmo puede expresarse en estas cuatro líneas de código TensorFlow, que calcula el *Squared Euclidean Distance*:

```
diff=tf.sub(expanded_vectors, expanded_centroides)
sqr= tf.square(diff)
distances = tf.reduce_sum(sqr, 2)
assignments = tf.argmin(distances, 0)
```

Y, si miramos las formas de los tensores, vemos que son respectivamente para *diff, sqr, distances* y *assignments*:

```
TensorShape([Dimension(4), Dimension(2000), Dimension(2)])
TensorShape([Dimension(4), Dimension(2000), Dimension(2)])
TensorShape([Dimension(4), Dimension(2000)])
TensorShape([Dimension(2000)])
```

Es decir, la función de *tf.sub* ha retornado el *tensor dist* que contiene la resta de los índices del centroide y del vector (indicado en la dimensión D1, centroide indicado en la dimensión D0, y por cada índice *x,y* indicado en la dimensión D2). El *tensor sqr* contiene el cuadrado de estos. En el *tensor distancia* vemos que ya ha reducido una dimensión, la indicada como parámetro en *tf.reduce_sum*.

Aprovecho este ejemplo para explicar que TensorFlow provee varias operaciones que se pueden usar para realizar operaciones matemáticas que reducen dimensiones de un *tensor* como es el caso de *tf.reduce_sum*. En la siguiente tabla pueden encontrar un resumen de las más importantes:

Operación	Descripción
tf.reduce_sum	Calcula la suma de los elementos a lo largo de una de las dimensiones
tf.reduce_prod	Calcula el producto de los elementos a lo largo de una de las dimensiones
tf.reduce_min	Calcula el mínimo de los elementos a lo largo de una dimensión
tf.reduce_max	Calcula el máximo de los elementos a lo largo de una dimensión
tf.reduce_mean	Calcula la media de los elementos a lo largo de una dimensión

Finalmente, la asignación se consigue con *tf.argmin*, que retorna el índice con el valor menor en la dimensión del *tensor* indicada (en nuestro caso la D0 que recordemos era la de los centroides).

También hay la operación *tf.argmax*:

Operación	Descripción
tf.argmin	Retorna el índice del elemento con el valor menor en la dimensión del *tensor* indicada
tf.argmax	Retorna el índice del elemento con el valor máximo en la dimensión del *tensor* indicada

En realidad, las 4 instrucciones anteriores se pueden resumir en una sola línea, tal como hemos presentado en el código de la sección anterior:

```
assignments = tf.argmin(tf.reduce_sum(tf.square(tf.sub(expanded_vectors, expanded_centroides)), 2), 0)
```

Pero de todas maneras, los *tensores* internos y las operaciones que se definen como nodos y ejecutan en el grafo interno son tal como lo hemos descrito en este apartado.

Cálculo de los nuevos centroides

Una vez hemos creado nuevos grupos en cada iteración, recordemos que el paso siguiente del algoritmo consiste en calcular los nuevos centroides de los nuevos grupos formados. En el código del apartado anterior se expresaba con esta línea de código:

```
means = tf.concat(0, [tf.reduce_mean(tf.gather(vectors, tf.reshape(tf.where( tf.equal(assignments, c)),[1,-1])), reduction_ind
```

En este código, se observa que el *tensor means* es el resultado de concatenar *k tensores* que corresponden al valor medio de todos los puntos pertenecientes a cada uno de los *k clusters*.

A continuación, comento cada una de las operaciones TensorFlow que intervienen en el cálculo del valor medio de los puntos pertenecientes a un *cluster*. Creo que el siguiente nivel de explicación es suficiente para el propósito de este libro:

- Con*equal* se obtiene un *tensor booleano* (*Dimension(2000)*) que indica (con valor "*true*") las posiciones donde el valor del *tensor assignment* coincide con el *cluster c* (uno de los *k*), del que en aquel momento estamos calculando el valor medio de sus puntos.
- Con *where* se construye un *tensor* (*Dimension (1) x Dimension(2000)*) con la posición donde se encuentran los valores "*true*" en el *tensor booleano* recibido como parámetro. Es decir, una lista con las posiciones de estos.
- Con *reshape* se construye un *tensor* (*Dimension (2000) x Dimension(1))* con los índices de los puntos en el *tensor vectors* que pertenecen a este *cluster c*.
- Con *gather* se construye un *tensor* (*Dimension (1) x Dimension (2000) x Dimension(2))* que reúne las coordenadas de los puntos que forman el *cluster c*.
- *Con tf.reduce_mean se* construye un *tensor* (*Dimension (1) x Dimension(2))* que contiene el valor medio de todos los puntos que pertenecen a este *cluster c*.

De todas formas, si el lector quiere profundizar un poco más en el código, como siempre indico, se puede encontrar más información para cada una de estas operaciones, con ejemplos muy aclaratorios, en la página de la API de TensorFlow[24]:

Ejecución del grafo

Finalmente, nos queda describir la parte del anterior código correspondiente al bucle y a la parte que actualiza los controides con los nuevos valores que contiene el *tensor means*. Para ello, tenemos que crear un operador que asigna el valor del *tensor means* a la variable *centroids* de tal manera que, cuando se ejecute la operación *run(),* en la próxima iteración del bucle se utilicen los valores de los centroides actualizados:

```
update_centroides = tf.assign(centroides, means)
```

También se tiene que crear un operador que inicialice todas las variables antes de empezar a ejecutar el grafo:

```
init_op = tf.initialize_all_variables()
```

Ahora ya está todo listo para que se pueda empezar a ejecutar el grafo:

```
sess = tf.Session()
sess.run(init_op)

for step in xrange(num_steps):
   _, centroid_values, assignment_values = sess.run([update_centroides, centroides, assignments])
```

En este código, por cada iteracción se actualizan los centroides y también la nueva asignación de *clusters* para cada uno de los puntos de entrada.

Fijémonos que el código especifica 3 operadores que tiene que ir a buscar en la ejecución de la llamada *run()*, y que se ejecutan en este mismo orden. Puesto que hay tres valores a buscar, *sess.run()* retornará una estructura de datos de tres elementos *numpy array* con el contenido del *tensor* correspondiente durante el proceso de entrenamiento.

Como *update_centroides* es una operación cuyo resultado no es el parámetro que retorna, el correspondiente elemento en la tupla que retorna no contiene nada, y por tanto será descartado, indicándolo con "_"[25]. Los otros dos valores, los centroides y la asignación de puntos a cada *clusters*, nos interesan para poder mostrarlos en pantalla una vez se han completado todos las *num_steps* iteracciones:

```
print centroid_values

[[ 2.99835277e+00 9.89548564e-01]
[ -8.30736756e-01 4.07433510e-01]
[ 7.49640584e-01 4.99431938e-01]
[ 1.83571398e-03 -9.78474259e-01]]
```

Espero que el lector tenga en pantalla unos similares, puesto que esto le indicará que ha podido ejecutar el código propuesto en este capítulo del libro.

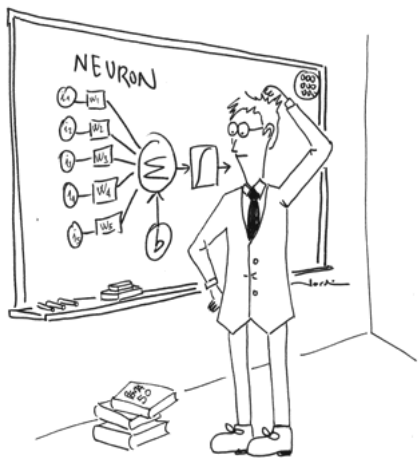
Le propongo al lector que antes de avanzar pruebe el código cambiando alguno los valores de *num_puntos* y especialmente K, y vea cómo cambia el resultado. Recuerden que para facilitar el probar el código descrito en este capítulo, este puede ser descargado del *github* del libro[26] con el nombre de *Kmeans.py*.

En este capítulo hemos avanzado en el conocimiento de TensorFlow, y en especial sobre la estructura de datos básica *tensor* a partir de un ejemplo de código en TensorFlow que implementa un algoritmo de *clustering K-means*.

Con estos conocimientos, ya estamos preparados para poder construir paso a paso una red neuronal sencilla de una capa con TensorFlow en el próximo capítulo.

[volver al índice de contenidos]

4. Red neuronal básica



En el prefacio, comentaba que una de las aplicaciones habituales de *Deep Learning* incluye reconocimiento de patrones. Por ello, de la misma manera que cuando uno empieza a programar existe la tradición de empezar por un *print "Hello World"*, en *Deep Learning* se empieza por crear un modelo de reconocimiento de números escritos a mano.

En este capítulo voy a presentar paso a paso cómo construir una red neuronal (*neural network*) sencilla de una sola capa en TensorFlow. Esta red neuronal va a reconocer dígitos escritos a mano, y es un ejemplo basado en uno de los diversos ejemplos del tutorial *beginners* de TensorFlow[27]. Dado el

carácter introductorio del libro, he optado por guiar al lector a través del ejemplo reduciendo todo lo que he sabido los conceptos y justificaciones teóricas de algunos de los pasos que se realizan.

Si el lector, después de leer este capítulo, le interesa profundizar más en los conceptos teóricos del ejemplo, le sugiero que continue con el libro disponible online *Neural Networks and Deep Learning*,**[28]** que en base al mismo ejemplo presenta con más profundidad los conceptos teóricos aquí usados.

Conjunto de datos MNIST

El conjunto de datos MNIST está formado por imágenes en blanco y negro de dígitos hechos a mano, que contiene 60.000 ejemplares para entrenar el modelo y 10.000 adicionales para testearlo. Los datos MNIST se pueden encontrar en la página *The MNIST database*[29].

Este conjunto de datos es ideal para la mayoría de personas que se adentran por primera vez en técnicas de reconocimiento de patrones en el mundo real sin tener que dedicar mucho tiempo al preproceso y formateado, ambos pasos muy importantes y costosos en tiempo cuando se está trabajando con imágenes.

Las imágenes originales en blanco y negro (*bilevel*) han sido normalizadas a 20×20 píxeles conservando su relación de aspecto. En este caso, es importante notar que las imágenes contienen niveles de grises como resultado de la técnica de *anti-aliasing*[30], usada en el algoritmo de normalización (reducir la resolución de todas las imágenes a una de más baja resolución). Posteriormente, las imágenes se han centrado en una de 28×28 píxeles, calculando el centro de masa de estos y trasladando la imagen con el fin de posicionar este punto en el centro del campo de 28×28. Las imágenes son del siguiente estilo:

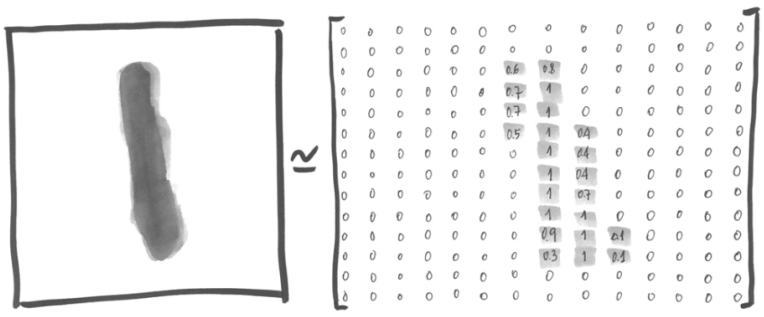
Además, recordemos que las imágenes incluyen una etiqueta que indica qué dígito representa, tratándose pues de un aprendizaje supervisado el que trataremos en este capítulo.

Para descargar los datos propongo que usen el código *input_data.py*[31] que, para su comodidad, he puesto en el *github* del libro (pero que es el obtenido de la página de Google[32]). Simplemente descarguen el código *input_data.py* en el mismo directorio de trabajo en que va a programar la red neuronal con TensorFlow. Desde el programa solo hace falta importarlo y usarlo de la siguiente manera:

```
import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Después de ejecutar estas dos instrucciones ya se dispone de los datos de entreno en *mnist.train* y los datos de test en *mint.test*. Como ya he dicho anteriormente, cada elemento está compuesto por una imagen, la cual referenciaremos como "*xs*", y su correspondiente etiqueta con notación "*ys*", para facilitar el expresar el código de procesado. Recordar que tanto los datos de entreno como los de test contienen "*xs*" y "*ys*". Por ejemplo, las imágenes de entreno están en *mnist.train.images* y las etiquetas de entreno están en *mnist.train.labels*.

Como ya he indicado, las imágenes constan de 28×28 píxeles, que se pueden representar como una matriz de números. Por ejemplo, una de las imágenes del número 1 se podría representar como:



Donde cada posición indica la intensidad del píxel entre 0 y 1. Esta matriz se puede representar con un *array* de 28×28 = 784 números. En realidad, la imagen se ha convertido en un montón de puntos en un espacio vectorial de 784 dimensiones. Solo mencionar que el reducir una estructura de 2D a este espacio estamos perdiendo parte de la información, y algunos algoritmos de visión por computador avanzados podrían afectar a su resultado, pero para el método más simple que usaremos en este tutorial esto no es un problema.

En resumen, tenemos un *tensor mnist.train.images* de 2D en el que si usamos la llamada *get_shape()* nos indica que tiene la forma *TensorShape([Dimension(60000), Dimension(784)]).* La primera dimensión indexa la imagen y la segunda indexa el píxel en cada imagen. Cada elemento en el *tensor* es la intensidad del píxel entre 0 y 1.

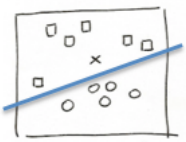
Por otro lado tenemos las etiquetas, que recordemos que son números entre 0 y 9 que indican qué dígito representa la imagen. En este ejemplo, vamos a representar esta etiqueta con un vector de 10 posiciones, donde la posición correspondiente al dígito que representa la imagen contiene un 1 y el resto son 0. Es decir, *mnist.train.labels* es un *tensor* con una forma de *TensorShape([Dimension(60000), Dimension10)]).*

Una neurona artificial

Aunque el enfoque del libro comporta el no entrar en los conceptos teóricos de las redes neuronales, creo que una breve explicación intuitiva de cómo funciona una neurona para cumplir con su cometido de aprender de los datos de entrenamiento puede ser de ayuda para el lector. Los lectores que conocen ya toda la teoría y solo buscan saber como usar TensorFlow pueden saltarse este apartado.

Veamos un ejemplo muy simple para ilustrar cómo puede aprender una neurona. Supongamos que tenemos un conjunto de puntos en un plano etiquetados ya como "cuadrado" o "redonda". Dado un nuevo punto "X", queremos saber qué etiqueta le corresponde:

Una aproximación habitual es dibujar una línea que separe los dos grupos y usarla como clasificador:



En este caso, los datos de entrada serán representados por vectores de la forma *(x,y)* que indican sus coordenadas en este espació de dos dimensiones, y nuestra función retornará '0' o '1' (encima o debajo de la línea) para saber si se debe clasificar como "cuadrado" o "círculo". Matemáticamente, en lo que habíamos ya aprendido en el capítulo de regresión lineal, "la línea" (el clasificador) puede ser definida por la recta $y = W * x + b$.

De manera más generalizada, una neurona, para clasificar valores de entrada X, debe aprender un vector de peso *W* (de la misma dimensión que los vectores de entrada X) y un sesgo *b* (llamado *bias* en redes neuronales). Con ellos, realizará una suma ponderada con el vector *W* de las entradas X, le sumará el sesgo *b*, y aplicará finalmente una función de "activación" no lineal para producir un resultado de '0' o '1'.

La función de esta neurona puede expresarse de una manera más formal, como:

$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1 & \text{si } z \geq 0 \\ 0 & \text{si } z < 0 \end{cases}$$

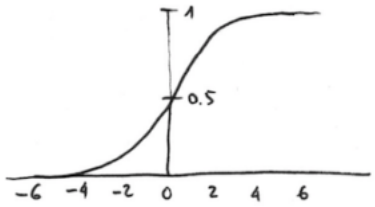
Definida esta función para la neurona, nos debemos ocupar de saber cómo la neurona puede aprender los parámetros *W* y *b* a partir de los datos que disponemos ya etiquetados como "cuadrados" o "redondas" en el ejemplo, para después poder saber cómo etiquetar este nuevo punto "X".

Una primera manera podría ser similar a lo que hicimos con la regresión lineal, es decir, alimentar esta neurona con los datos de entrada conocidos, y comparar el valor obtenido y el real. Después de cada iteración, los pesos de *W* y *b* son ajustados de tal manera que se minimice el error, como hicimos en el capítulo 2 para la recta de la regresión lineal.

Una vez tenemos los parámetros *W* y *b* podemos calcular la suma ponderada, y entonces nos hace falta una función que aplique una transformación para que se convierta en '0' o '1' el resultado almacenado en *z* . Aunque hay varias funciones de activación, pare este ejemplo podríamos usar una bastante popular, que es la función *sigmoid*[33], que retorna un valor real de salida entre 0 y 1:

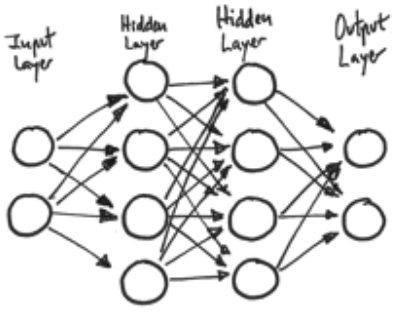
$$z = b + \sum_i x_i w_i \qquad y = \frac{1}{1+e^{-z}}$$

Si se piensa un poco en la fórmula, veremos que tiende siempre a dar valores próximos al 0 o al 1. Si la entrada *z* es razonablemente grande y positiva, "e" a la menos z es cero y, por tanto, la *y* es 1. Si la entrada es grande y negativa, "e" elevado a un número positivo grande y el denominador resultará ser un número grande y por lo tanto la salida *y* será 0. Gráficamente, la función queda de esta forma:



Hasta aquí hemos presentado como se puede definir una neurona, pero una red neuronal es una composición de neuronas conectadas entre sí de diferentes maneras y usando diferentes funciones de activación. Dado el alcance de este libro, no podemos entrar a describir toda la extensión del universo de las redes neuronales, pero les aseguro que es emocionante.

En todo caso, solo mencionar un caso concreto de redes neuronales (en el que basaremos el capítulo 5) donde las neuronas están organizadas en capas de tal manera que las de la capa inferior (*input layer*) reciben señales de las entradas, y las neuronas en la capa superior dan la respuesta (*output layer*). La red neuronal puede tener varias capas intermedias llamadas *hidden layers*. Una forma visual de representarlo es:



En estas redes, las neuronas de una capa se comunican con las que pertenecen a una capa superior para recibir información, y estas a su vez comunican su información a las neuronas de la capa inferior.

Como ya hemos dicho, hay varias funciones de activación además de la *Sigmoid*, cada una con sus propiedades diferentes. Por ejemplo, cuando se quiere clasificar más de dos clases en el *output layer*, se usan habitualmente otras funciones de activación como pueden ser la función *Softmax*[34] que, en cierta manera, es una generalización de la función *Sigmoid*. *Softmax* permite obtener la probabilidad para cada clase, de tal manera que la suma de todas ellas sea 1.

Un modelo sencillo para empezar: Softmax

Recordemos que nuestro problema a resolver es ser capaces de, dada una imagen de entrada, obtener las probabilidades de que sea cada uno de los 10 posibles dígito. Por ejemplo, nuestro modelo podría predecir en una imagen un nueve y estar seguro en un 80% de que es un nueve, pero asignar un 5% de posibilidades de que sea un ocho (debido al dudoso bucle inferior) y asignar una cierta probabilidad de poder ser cualquier otro número. Esto es debido a una cierta incertidumbre por no ser seguro, es decir, no podemos reconocer los dígitos con un 100% de confianza (*confidence*). En este caso, usar una distribución de probabilidades nos puede dar una mejor idea de cuánto de confiados estamos de nuestra predicción.

Por tanto, nos hace falta como vector de salida una distribución de probabilidad sobre un conjunto de etiquetas mutuamente excluyentes. Es decir, en nuestro vector de 10 probabilidades cada una correspondiente a un dígito y que todas ellas sumen 1.

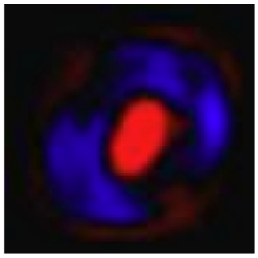
Como ya he avanzado, esto se logra mediante el uso de una capa de salida con una función de activación *softmax*. La salida de una neurona en una capa *softmax* depende de las salidas de todas las otras neuronas en su capa, puesto que la suma de todas ellas debe ser 1, como hemos dicho.

La función softmax se concreta en dos pasos principales: primero calculamos "las evidencias" de que una determinada imagen pertenece a una clase en particular y después convertimos estas evidencias en probabilidades de que pertenezca a cada una de las 10 clases posibles.

Evidencia de pertenencia

Para medir la evidencia de que una determinada imagen pertenece a una clase en particular, una aproximación muy usada consiste en realizar una suma ponderada de las intensidades de los píxeles. El peso es negativo si ese píxel que tiene una alta intensidad se evidencia en contra de estar en esa clase, y positivo si se evidencia a favor.

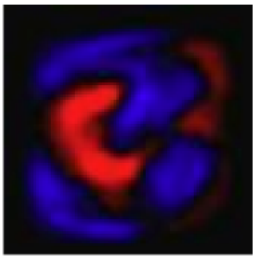
Para explicarlo con un ejemplo gráfico; supongamos que disponemos del modelo aprendido para el número cero (más adelante veremos cómo se aprenden estos modelos). Por el momento, podemos ver un modelo como "algo" que contiene información para saber si un número es de una determinada clase. En este caso, hemos escogido un modelo como el que presentamos a continuación, donde el rojo (en la edición en blanco/negro del libro es el gris más claro) representa pesos negativos (es decir, reducir la evidencia de que pertenece), mientras que el azul (en la edición en blanco/negro del libro es el gris más oscuro) representa los pesos positivos. Mírenlo atentamente:



Imaginemos una hoja en blanco de 28×28 píxeles y tracemos un cero en ella. En general, el trazo de nuestro cero caería sobre la zona azul (recordemos que las imágenes habían sido normalizadas a 20×20 píxeles y posteriormente centradas a una imagen de 28×28).

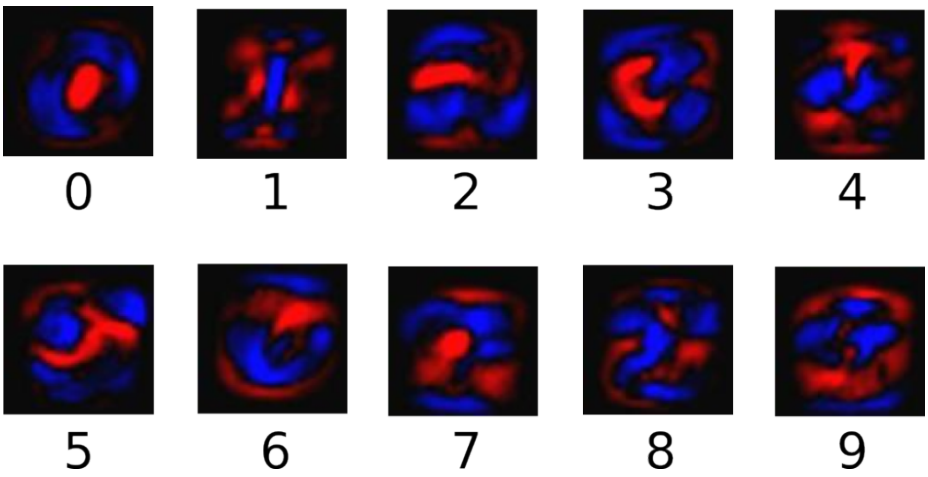
Resulta bastante evidente que si nuestro trazo pasa por encima de la zona roja lo más probable es que no estemos trazando un cero. Por tanto, usar una métrica basada en sumar si trazamos en zona azul y restar si pasa por zona roja puede parecer razonable.

Imaginemos que trazamos un tres; está claro que la zona roja del centro del anterior modelo de referencia va a penalizar la métrica antes mencionada. Pero en cambio, si el modelo de referencia es el siguiente, podemos observar que, en general, los diferentes posibles trazos que representan un tres se mantienen en la zona azul. En este caso, el trazado de un cero quedaría en la parte de la zona roja.



Espero que el lector, viendo estos dos casos concretos, pueda visualizar como la aproximación de los pesos indicados anteriormente nos permite hacer una estimación de qué número se trata.

En la siguiente página se muestra los pesos de un ejemplo concreto de modelo aprendido para cada una de estas diez clases del MNIST (extraído del ejemplo de Tensorflow[35]). Recordemos que hemos escogido el rojo (gris más claro) en esta representación visual para los pesos negativos, mientras que usaremos el azul (gris más oscuro) para representar los positivos.



De manera más formal, podríamos expresar que la evidencia para una clase *i* dado un input *x* se puede calcular con la siguiente expresión:

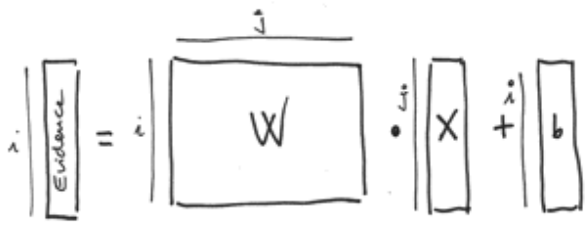
$$evidence_i = \sum_j W_{i,j} x_j$$

Donde *i* indica la clase (en nuestro caso entre 0 y 9), *j* es un índice para sumar a lo largo de los píxeles de nuestra imagen de entrada. Finalmente, *Wi* representa los pesos anteriormente mencionados.

Recordemos que, en general, los modelos incluyen un parámetro extra que representa el sesgo (*bias* en inglés), indicando un cierto margen de incertidumbre. En nuestro caso la fórmula final quedaría como:

$$evidence_i = \sum_j W_{i,j} x_j + b_i$$

En la que por cada *i* (entre 0 y 9) tenemos una matriz *Wi* de 784 elementos. (28×28), donde cada elemento *j* de la matrix se multiplica por el correspondiente componente *j* de la imagen de entrada (de 784 componentes) y se le suma el b_i correspondiente. Una visión grafica del cálculo matricial y los índices de recorrido son:



Probabilidades de pertenencia

Hemos comentado que el segundo paso consistía en calcular unas probabilidades. En concreto, se trata de convertir el recuento de evidencias en probabilidades predichas, que indicaremos con la variable *y*, usando la función *softmax*:

$$y = \text{softmax}(\text{evidence})$$

Recordemos que el vector de salida debe ser una distribución de probabilidad cuya suma de todos sus componentes sume 1. Para conseguir normalizar cada componente de manera que sume 1, la función *softmax* usa el valor exponencial de sus entradas y luego las normaliza como se muestra en la siguiente expresión:

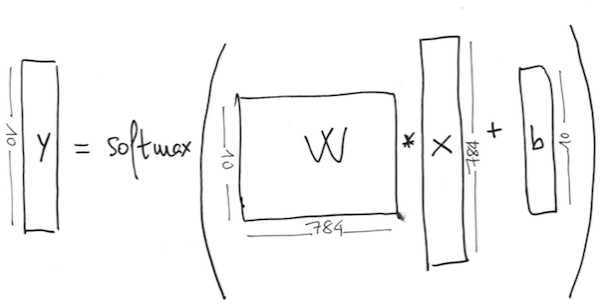
$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

El efecto que se consigue con el uso de exponenciales es que una unidad más de evidencia tiene un efecto multiplicador en el peso. Y a la inversa, una unidad menor de evidencia significa que una hipótesis obtiene una fracción de su peso anterior. A su vez, *softmax* normaliza estos pesos de modo que sumen uno, formando una distribución de probabilidad.

Lo interesante de esta función es que una buena predicción tendrá una sola entrada en el vector con valor cercano a 1, mientras que las entradas restantes estarán cerca de 0. En una predicción débil tendrán varias etiquetas posibles, que tendrán más o menos la misma probabilidad.

Programa en TensorFlow

Después de esta breve descripción de qué debería hacer el algoritmo para poder reconocer los dígitos, pasemos a implementarlo en TensorFlow. Para ello, nos puede ir bien hacer un breve repaso de cómo deben ser los *tensores* que almacenan nuestros datos y los parámetros del modelo. Para este propósito me he permito hacer el siguiente esquema, que visualiza las estructuras de datos y la relación entre ellas (espero que sea de utilidad para el lector):



Para empezar, debemos crear dos variables que contengan los pesos *W* y sesgos *b* del modelo.

```
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))
```

Se crean estas variables dando a *tf.Variable* el valor inicial de la variable, que en este caso se inicia con un *tensor* constante inicializado a su vez a ceros.

Debemos fijarnos que *W* tiene una forma *TensorShape ([Dimension(784), Dimension(10)])* que se define gracias al argumento que recibe, que es el de un *tensor* de constantes con esta forma. Lo mismo ocurre con la variable de sesgo *b*, que obtiene la forma del argumento que recibe (*TensorShape ([Dimension(10)])*).

La matriz *W* tiene este tamaño porque queremos multiplicar el vector de imagen de tamaño 784 por cada uno de los 10 dígitos para producir un *tensor* de evidencias después de sumarle *b*.

En nuestro caso de estudio MNIST, también crearemos un *tensor* de 2 dimensiones para tener la información de los puntos *x* mediante la siguiente línea de código:

```
x = tf.placeholder("float", [None, 784])
```

El *tensor x* se usará para guardar imágenes MNIST en forma de vector de 784 de números de coma flotante (con *None* indicamos que la dimensión puede ser de cualquier tamaño; en nuestro caso será igual al número de elementos que incluyamos en el proceso de aprendizaje).

Ahora que ya tenemos definidos los *tensores*, podemos implementar nuestro modelo. Para ello, TensorFlow provee varias operaciones, siendo una de ellas *tf.nn.softmax(logits, name=None),* que implementa la función que se ha presentado en la anterior sección. El argumento debe ser un *tensor* y, opcionalmente, un nombre. La función retorna un *tensor* con el mismo tipo y forma que el *tensor* pasado como argumento.

En nuestro caso, pasamos a esta función como argumento el *tensor* resultado de la multiplicación de la imagen vectorizada *x* por la matriz de pesos *W* y sumándole el parámetro *b* :

```
y = tf.nn.softmax(tf.matmul(x,W) + b)
```

Una vez tenemos la implementación del modelo podemos pasar a especificar el código necesario para encontrar los pesos *W* y sesgos *b* de la red mediante el algoritmo iterativo de entrenamiento. En cada iteración, el algoritmo de *training* coge los datos de entrenamiento, aplica la red neuronal y compara el resultado obtenido con el esperado.

Con el fin de entrenar a nuestro modelo y saber cuándo tenemos uno bueno, tenemos que definir de alguna manera lo que significa que el modelo sea "bueno". Como ya vimos en anteriores capítulos, lo que se hace habitualmente es definir lo contrario, es decir, calcular lo "malo" que es ese modelo con funciones de coste (*cost*). En este caso, el objetivo es intentar conseguir valores de los parámetros W y b que minimicen el valor de la métrica que indica cuan malo es el modelo.

Hay diferentes métricas del grado de error entre salidas calculadas y las salidas deseadas de los datos de entrenamiento. Una medida común de error es el *mean squared error* o el *Squared Euclidean Distance*, ambos vistos anteriormente. Sin embargo, hay algunos resultados de investigación que sugieren utilizar otras métricas diferentes para una red neuronal como esta. Un ejemplo puede ser el llamado error de entropía cruzada (*cross entropy error*), y es el que usaremos en nuestro ejemplo. Esta métrica se define como:

$$-\sum_i y_i' \log(y_i)$$

Donde *y* es la distribución de probabilidad precedida y la *y'* la distribución real (obtenida a partir del etiquetado de los datos de entrada). Puesto que la matemática detrás de *cross-entropy* y su papel en redes neuronales es bastante compleja, no entraré en más detalle, en todo caso dejar solo la idea intuitiva de que el mínimo se consigue cuando las dos distribuciones de probabilidades son las misma; nuevamente remito al lector al libro *Neural Networks and Deep Learning*,[36] disponible online si está interesado en más detalles.

Para implementar el *cross-entropy* necesitamos un nuevo *placeholder* para entrar la respuesta correcta:

```
y_ = tf.placeholder("float", [None,10])
```

A partir de este *placeholder* podemos implementar el *cross-entropy* con la siguiente línea de código, que representará nuestra función de coste:

```
cross_entropy = -tf.reduce_sum(y_*tf.log(y))
```

Primero, se calcula el logaritmo de cada elemento *y* con la función que nos provee TensorFlow *tf.log* y después se multiplica por cada elemento *y_* . Finalmente, con *tf.reduce_sum* se suman todos los elementos del *tensor* (más adelante veremos que las imágenes se miran por paquetes, y en este caso el valor de *cross-entropy* corresponde a la suma de las *cross-entropy* de las imágenes de un paquete y no a la de una sola).

De forma iterativa, una vez determinado el error para una muestra se trata de corregir el modelo (modificar los parámetros *W* y *b* en nuestro caso) para reducir la diferencia entre valores calculados y esperados en la próxima iteración con otro valor de entrada.

Hay varios algoritmos, pero en nuestro ejemplo usaremos el conocido como *backpropagation,* abreviación de *backward propagation of errors,* que como su nombre indica, propaga hacia atrás el error para poder recalcular los pesos *W.* Este método se usa conjuntamente con otro de optimización como puede ser el *gradient descent* ya visto anteriormente, que mediante el cálculo del gradiente de la función de coste elegida *(*en nuestro caso la *cross-entropy)* nos permite ir calculando cómo ir cambiando los parámetros para reducir el error en cada iteración de acuerdo a la información local que se dispone en ese momento.

Dado el carácter introductorio de este tutorial no entraremos en detalle en estos métodos, pero intuitivamente consiste en cambiar el valor de los pesos *W* un poquito en cada iteración (expresando con un parámetro *learning rate* la velocidad con que lo cambiamos) para reducir la función de error. Recordemos que TensorFlow conoce todo el grafo de computación, y esto le permite aplicar algoritmos de optimización para poder encontrar los gradientes de función de coste que le permite entrenar los modelos automáticamente.

Para nuestro ejemplo de imágenes de MNIST, el siguiente comando indica a TensorFlow que use el algoritmo de *backpropagation* para minimizar el *cross-entropy* usando el algoritmo *gradient descent* (con un l*earning rate* de 0.01):

```
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
```

Hasta aquí, hemos realizado toda la especificación y ahora ya podemos empezar a computar instanciando a la clase *tf.Session()* que ejecuta las operaciones TensorFlow entre los dispositivos (*Devices*) CPUs o GPUs disponibles en el sistema:

```
sess = tf.Session()
```

A continuación podemos ejecutar la primera operación que inicializa las variables:

```
sess.run(tf.initialize_all_variables())
```

A partir de este momento ya podemos empezar a entrenar nuestro modelo. El parámetro retornado por *train_step*, cuando se ejecute, aplicará la actualización del *gradient descent* a los parámetros. Por tanto, entrenar el modelo puede ser conseguido repitiendo la ejecución del *train_step*.

Por ejemplo, supongamos que lo queremos ejecutar 1.000 veces, el código podría ser el siguiente:

```
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

En la primera línea del bucle se especifica que en cada iteración de entrenamiento se coge un lote de 100 datos escogidos al azar del conjunto de datos de entreno. Podríamos usar todos los datos en cada iteración pero, para hacer más rápido este primer programa, usamos solo un pequeño conjunto en cada iteración. En la segunda línea se indica que se ejecute *train_step,* indicándole que los datos leídos anteriormente alimenten a los respectivos *placeholders*.

Como nota final de esta sección, mencionar que los algoritmos de *Machine Learning* basados en gradiente se benefician de la capacidad de diferenciación automática de TensorFlow. Como usuario TensorFlow, solo se tiene que definir la arquitectura computacional del modelo predictivo, combinarla con la función objetivo, y simplemente añadir los datos. TensorFlow se encarga de manejar el cálculo de derivadas asociado que hay detrás de todo el proceso de aprendizaje.

Evaluación del modelo

Es importante evaluar cuan bueno es un modelo. Por ejemplo, podemos calcular el porcentaje de aciertos averiguando dónde predijimos la etiqueta correcta. Recordemos de capítulos anteriores que *tf.argmax* es una función que retorna el índice de la entrada más alta en un *tensor* a lo largo de un eje. Por ejemplo, *tf.argmax (y, 1)* es la etiqueta de nuestro modelo que es más probable para cada entrada, mientras que *tf.argmax (y_, 1)* es la etiqueta correcta. Junto con *tf.equal* podemos comprobar si nuestra predicción coincide con la verdadera de la siguiente manera:

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

Esta instrucción nos retorna una lista de *booleanos*. Para determinar qué fracción es correcta, podemos pasar los números a coma flotante y entonces calcular la media con la siguiente

instrucción:

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

Por ejemplo, *[True, False, True, True]* se convertirán en *[1,0,1,1]* donde la media sale 0.75 y que representa el porcentaje de exactitud (*accuracy*). Ahora, podemos pedir la *accuracy* de nuestros datos de test usando *mnist.test* en el argumento de *feed_dict*:

```
print sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels})
```

A mí me ha salido un valor alrededor del 91% por pantalla. ¿Son buenos estos resultados? Yo creo que son fantásticos, porque esto significa que el lector ya ha programado y ejecutado su primera red neuronal con TensorFlow. Otra cosa es que haya modelos que permiten mejorar la *accuracy* mucho más. Y esto es lo que presentaremos en el siguiente capítulo con una red neuronal con más capas.

El lector puede encontrar el código que hemos usado en este capítulo en el fichero *RedNeuronalSimple.py* en el *github* del libro[37], aunque lo incluyo a continuación para facilitar su estudio con una visión global:

```
import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

import tensorflow as tf

x = tf.placeholder("float", [None, 784])
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))

matm=tf.matmul(x,W)
y = tf.nn.softmax(tf.matmul(x,W) + b)
y_ = tf.placeholder("float", [None,10])

cross_entropy = -tf.reduce_sum(y_*tf.log(y))
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)

sess = tf.Session()
sess.run(tf.initialize_all_variables())

for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels})
```

[volver al índice de contenidos]

5. Red neuronal Deep Learning

En este capítulo programaré, junto con el lector, una red *Deep learning* simple usando el mismo problema de reconocimiento de dígitos del MNIST del capítulo anterior.

Como ya he avanzado, una red neuronal *Deep learning* la conforman varias capas apiladas una encima de la otra. En concreto, en este capítulo programaremos una red neuronal convolucional, un caso concreto de redes neuronales *Deep learning*, que fueron popularizadas por Yann LeCunn, entre otros, ya en el 1998, pero que en estos últimos años han conseguido resultados muy interesantes en el reconocimiento de imagen; por ejemplo: en nuestro caso de reconocimiento de dígitos se consigue una exactitud de más del 99%.

Nuevamente, en este capítulo voy a usar un ejemplo de código como elemento vertebrador y a partir de él introduciré dos de lo conceptos más relevantes en el tema como son convolucional y *pooling*, aunque sin entrar en detalles de parámetros, dado el carácter introductorio del libro. No obstante, el lector podrá poner en funcionamiento el código, y espero que esto le permita llegar a

comprender la idea global de las redes convolucionales.

Redes neuronales convolucionales

Las redes neuronales convolucionales (*Convolutional Neural Networks* en inglés, con los acrónimos CNNs o *ConvNets*), son un caso particular de *Deep learning* y han impactado profundamente en el área de visión por computador.

Pero un rasgo diferencial de las CNN es que hacen la suposición explícita de que las entradas son imágenes, cosa que nos permite codificar ciertas propiedades en la arquitectura. Esto permite una implementación más eficiente y la reducción de parámetros requeridos en la CNN. Vamos a verlo a través de nuestro ejemplo de reconocimiento de dígitos MNIST: después de cargar los datos en MNIST y definir los *placeholders* como en el ejemplo anterior con las siguientes instrucciones TensorFlow:

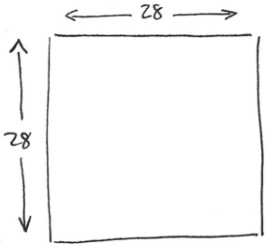
```
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

import tensorflow as tf

x = tf.placeholder("float", shape=[None, 784])
y_ = tf.placeholder("float", shape=[None, 10])
```

Podemos devolver la forma de imágenes a los datos de entrada. Esto lo hacemos con:

```
x_image = tf.reshape(x, [-1,28,28,1])
```

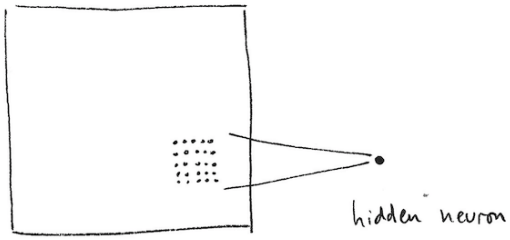
Donde cambiamos la forma de las imágenes a un *tensor* 4D, con la segunda y tercera dimensión correspondiendo a la anchura y la altura de la imagen, mientras que la última dimensión corresponde al número de canales de color (en nuestro caso 1).

Así pues, como entrada en nuestra red neuronal podemos pensar en un espacio de neuronas de dos dimensiones y tamaño 28×28 como las de la figura:



Hay dos primitivas principales que definen a las redes convolucionales: los filtros y los mapas de características. Estas primitivas pueden expresarse como grupos de neuronas especializadas, como veremos a continuación. Pero primero vamos a hacer una breve descripción de estas dos primitivas, dado su relevante papel en las CNN.

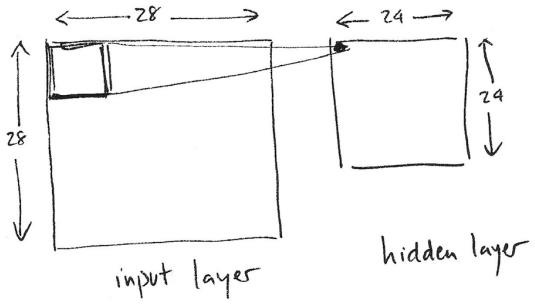
De manera intuitiva, podríamos decir que el propósito principal de una capa convolucional es detectar características o rasgos visuales en las imágenes como aristas, líneas, gotas de color, etc. De ello se encarga una primera capa de neuronas ocultas (*hidden*) que están conectadas a las neuronas de la capa de entrada que hemos comentado. Pero en el caso de CNN que nos ocupa, no se conectan todas las neuronas de entrada con todas las neuronas de este primer nivel de neuronas ocultas; solo se hace por pequeñas zonas localizadas del espacio de las neuronas de entrada que almacenan los píxeles de la imagen. Visualmente, se podría representar como:



Para ser más precisos, en el ejemplo anterior cada neurona de nuestra capa oculta será conectada a una pequeña región de 5×5 neuronas (es decir 25 neuronas) de la capa de entrada.

Intuitivamente, se puede pensar en una ventana del tamaño de 5×5 que va recorriendo toda la capa de 28×28 de entrada que contiene la imagen. Esta ventana va deslizándose a lo largo de toda la capa de neuronas, en realidad superponiéndose entre ellas estas ventanas. Por cada posición de la ventana hay una neurona en la capa oculta que procesa esta información.

Visualmente, podemos suponer que empezamos con la ventana en la zona arriba-izquierda de la imagen, y esto le da la información necesaria a la primera neurona de la capa oculta. Después, deslizamos la ventana una posición hacia la derecha para "conectar" estas 5×5 neuronas de la capa de entrada con la segunda neurona de la capa oculta. Y así, sucesivamente, vamos recorriendo todo el espacio de la capa de entrada, de izquierda a derecha y de arriba a abajo:



Analizando un poco el caso concreto que hemos propuesto, observemos que si tenemos una entrada de 28×28 píxeles y una ventana de 5×5 esto nos define un espacio de 24×24 neuronas en la primera capa del nivel oculto, debido a que solo podemos mover a la ventana 23 neuronas hacia la derecha y 23 hacia abajo antes de chocar con el lado derecho (o inferior) de la imagen de entrada. Ahora bien, esto según el supuesto de que la ventana hace movimientos de avance de 1 píxel de distancia, donde en cada paso la nueva ventana se solapa con la anterior excepto en esta línea de píxeles que hemos avanzado.

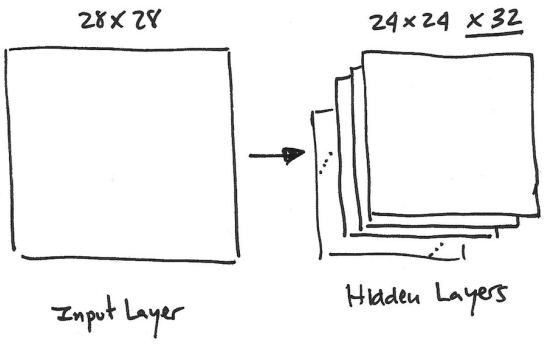
Pero en CNN se pueden usar diferents longitudes de pasos de avance, mediante el parámetro llamado *stride length.* En las CNN también se puede aplicar una técnica de relleno de ceros alrededor del margen de la imagen para mejorar el barrido que se realiza con ventana que se va deslizando. El parámetro para definir este relleno recibe el nombre de *padding*[38], con el que se puede especificar el tamaño de este relleno, pero en nuestro ejemplo no entraremos a analizar en detalle en estos parámetros dado el ámbito de este libro.

En nuestro caso de estudio, y siguiendo el formalismo del anterior capítulo, para "conectar" cada neurona de la capa oculta con las 25 neuronas que le corresponden de la capa de entrada usaremos un valor de sesgo b y una matriz de pesos W de tamaño 5×5. Ahora bien, lo particular y muy importante de las redes convolucionales es que se usa la misma matriz W de pesos y el mismo sesgo b para todas las neuronas de la capa oculta, en nuestro caso para las 24×24 neuronas (576 neuronas). El lector puede ver claramente que esta compartición reduce de manera drástica el número de parámetros que tendría una red neuronal si no la hiciéramos. Es decir, pasa de 14.400 parámetros (5x5x24x24) a 25 (5×5) parámetros.

Esta matriz compartida junto con el sesgo se acostumbra a llamar *kernel* o *filter* en este contexto de redes convolucionales. En realidad, si el lector está familiarizado con programas de tratamiento de imágenes, es como los filtros que usamos para retocar las imágenes, que en nuestro caso sirven para buscar características locales en pequeños grupos de entradas. Les recomiendo ver los

ejemplos que se encuentran en el manual del editor de imágenes GIMP[39] para hacerse una idea visual y muy intuitiva de cómo funciona un proceso de convolución.

Pero un *kernel* definido por una matriz y un sesgo solo permite detectar una característica concreta en una imagen, por tanto, para poder realizar reconocimiento de imágenes se propone usar varios *kernels* a la vez, uno para cada característica que queramos detectar. Por eso una capa convolucional completa en una CNN comprende varios *kernels*. Una manera habitual de representar visualmente esta capa convolucional es la siguiente:



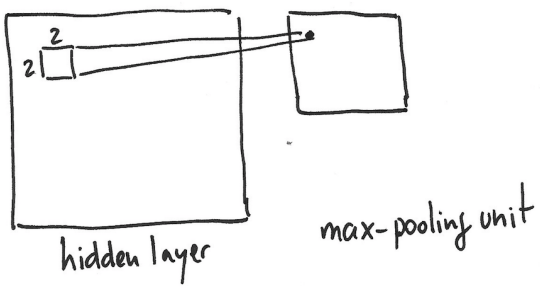
Donde el primer nivel de capas ocultar está compuesta por varios *kernels*. En nuestro ejemplo, usaremos 32 *kernels*, donde cada *kernel* se define con una matriz *W* de pesos compartida de 5×5 y un sesgo *b*, también compartido entre las neuronas que conforman este *kernel*.

Para simplificar el código propongo definir las siguientes dos funciones puesto que se requieren múltiples *W* y *b* para una sola capa, y esto facilitará la especificación de nuestro código:

```
def weight_variable(shape):
initial = tf.truncated_normal(shape, stddev=0.1)
return tf.Variable(initial)

def bias_variable(shape):
initial = tf.constant(0.1, shape=shape)
return tf.Variable(initial)
```

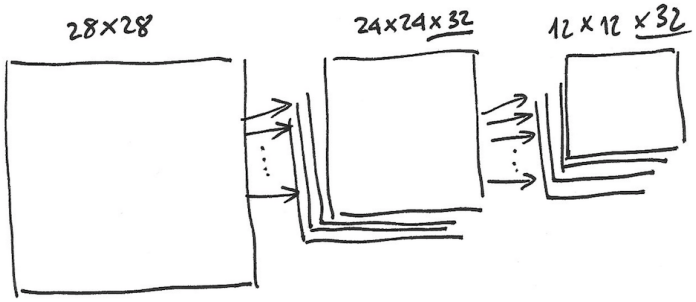
Sin entrar en detalle, diremos que se acostumbra a inicializar los pesos con un poco de ruido y con valores ligeramente positivos los sesgos *b*, que es lo que hemos hecho en este caso.

Además de las capas convolucionales que acabamos de describir, las redes neuronales convolucionales acompañan a la capa de convolución con unas capas llamada *pooling*, que suelen ser utilizada inmediatamente después de las convolucionales. Lo que las capas de *pooling* hacen es simplificar la información recogida por la capa convolucional y crear una versión condensada de las características. En nuestro ejemplo, vamos a tomar una región de 2×2 de la capa convolucional y vamos a sintetizar la información en un punto en la capa de *pooling*.



Hay varias maneras de condensar la información, pero una habitual, y que usaremos en nuestro ejemplo, es la conocida como *max-pooling*, que como valor se queda con el valor máximo de los que había en la entrada de 2×2 en nuestro caso.

Tal como hemos mencionado anteriormente,la capa convolucional alberga más de un kernel, y por tanto, como aplicamos el *max-pooling* a cada uno de ellos separadamente, habrá tantas capas de *pooling* como de convolución:



El resultado es que dado que teníamos un espacio de 24×24 neuronas en la capa convolucional, después de hacer el *pooling* tenemos 12×12 neuronas que corresponde a las 12×12 *tiles* de 2×2 aparecidas al dividir el espacio de *tiles* disjuntas (notar que aquí se divide el espacio en *tiles* del tamaño de la ventana, no como antes, que se desplazaba la ventana) .

De manera intuitiva, podemos explicar que *max-pooling* es una manera de saber si una determinada característica se encuentra en cualquier lugar de la imagen, y una vez que se ha encontrado una característica, su ubicación exacta no es tan importante como su ubicación aproximada en relación con otras características.

Implementación del modelo

A partir de aquí voy a presentar el código, ejemplo de cómo se puede programar una CNN basada en uno de los ejemplos avanzados (*Deep MNIST for experts*) que se encuentran en la página web de TensorFlow[40] . Como ya dije al principio, dado que hay muchos detalles de los parámetros que requieren una explicación teórica adicional y más detallada que la que permite este libro, me limitaré a dar una visión global del código sin entrar en muchos de los detalles que se derivan de los parámetros de las funciones de TensorFlow.

Como hemos visto, hay varios valores a concretar para parametrizar las etapas de convolución y *pooling*. En nuestro caso, usaremos un modelo simplificado con un *stride* de 1 en cada dimensión (tamaño del paso con el que desliza la ventana) y un *padding* de 0 (relleno de ceros alrededor de la imagen). El *pooling* será un *max-pooling* como el descrito sobre unos bloques de 2×2. Nuevamente, para generar un código más ordenado, propongo estas dos funciones genéricas para las dos transformaciones que describen la convolución y el *pooling* :

```
def conv2d(x, W):
 return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')


def max_pool_2x2(x):
 return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

Pasemos a implementar nuestra primera capa, que consistirá en una convolución seguida del *max-pooling*. En nuestro caso, tendremos 32 características usando una ventana de 5×5. Esto nos lleva a que debemos definir un *tensor* para la matriz de pesos *W* con forma [5, 5, 1, 32]. Las dos primeras dimensiones son el tamaño de la ventana, el siguiente es el número de canales (en nuestro caso, 1) y el último corresponde al número de características que hemos decidido tener. Como es evidente, también tendremos un valor de sesgo para cada una de las 32 características. El código en TensorFlow que lo especifica, teniendo en cuenta las anteriores funciones definidas, es:

```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

La función *ReLU* (*Rectified linear unit*) se ha convertido ultimamente en la función de activación habitual en las capas ocultas de una *deep neural network*. Básicamente, la función consiste en devolver el valor *max*(0,*x*), es decir, para valores negativos retorna un cero. En nuestro ejemplo, aplicaremos esta función de activación con el valor que nos retorna la función de convolución.

El código que generaremos consiste primero en aplicar la función de convolución a las imágenes *x_image,* que retorna el resultado de hacer un barrido del filtro 2D indicado por el *tensor* de pesos

W_conv1, y sumarle el sesgo antes de pasar este valor a la función *ReLU*. Finalmente, a este resultado se le aplica la función *max-pool*:

```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)

h_pool1 = max_pool_2x2(h_conv1)
```

Con el fin de construir una *deep neural network*, podemos apilar varias capas como la construida hasta aquí. Para mostrar al lector como hacerlo en nuestro ejemplo, crearé una segunda capa que tendrá 64 características para una ventana de 5×5. En este caso, el parámetro de número de canales de entrada tomará el valor de las 32 características que hemos obtenido de la capa anterior:

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

El tamaño de la capa de convolución resultante es 7×7 dado que ahora partimos de un espacio de entrada de 12×12 y una ventana deslizante de 5×5, teniendo en cuenta que tiene un *stride* de 1.

El siguiente paso, ahora que el tamaño de la imagen se ha reducido a 7×7, consiste en añadir una capa totalmente conectada (*densely connected layer*), que servirá para alimentar una capa final de *softmax* como la del capítulo anterior.

Usaremos una capa con 1024 neuronas para permitir el procesamiento de la imagen entera. Los *tensores* para los pesos y el *bias* serán:

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
```

Recordemos que el tamaño de la primera dimensión del *tensor* ha de representar las 64 características de 7×7 resultante de la segunda capa de convolución, y el segundo parámetro es el número de neuronas que tiene la capa, que es arbitrario el valor, podría ser otro.

Ahora volvemos a cambiar el *tensor* a una forma más aplanada, de vector. Recordemos del anterior capítulo que *softmax* procesa imágenes aplanadas en forma de vector. Esto se consigue multiplicando por la matriz de pesos *W_fc1*, añadimos el sesgo *b_fc1* . Finalmente aplicamos la función *ReLU* nuevamente:

```
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])

h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

El siguiente paso será reducir el número de parámetros del modelo, mediante lo que en redes neuronales se conoce como *dropout*, que consiste en eliminar neuronas y todos sus arcos de entrada y salida. La decisión de qué neuronas eliminar es random. Una manera de hacerlo consiste en aplicar una probabilidad, como veremos en nuestro código.

Sin entrar en mucho detalle, diré que este paso se realiza para reducir el riesgo que esto comporta de sobreajustar el modelo cuando tenemos capas ocultas que generan modelos muy expresivos, y puede llegar a pasar que parte del ruido (o error) *random* queda modelizado. Es lo que se conoce como *overfitting*, que puede aparecer cuando un modelo tiene muchos parámetros en relación al número de valores de entrada. Esta situación se intenta evitar, puesto que genera un pobre rendimiento predictivo.

En nuestro caso de estudio lo aplicaremos, y consiste en usar la función de *dropout tf.nn.dropout* en TensorFlow antes de la última capa *softmax*. Para ello, en nuestro ejemplo crearemos un *placeholder* para almacenar la probabilidad de que una salida de una neurona se mantenga durante el proceso de *dropout*:

```
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Finalmente, añadimos a nuestro modelo una capa *softmax* como la que ya vimos en el anterior capítulo, que recordemos que permite obtener la probabilidad para cada clase (dígito en nuestro

caso) de tal manera que la suma de todas ellas sea 1. El código de nuestro ejemplo para especificar esta capa *sofmax* es el siguiente:

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

Entrenamiento y evaluación del modelo

Estamos ya en disposición de pasar a entrenar el algoritmo, es decir, ajustar los parámetros de todas las capas convolucionales para obtener el resultado deseado a partir de las imágenes que tenemos etiquetadas. A partir de aquí, para saber cuán bien lo hace nuestro modelo, debemos hacer lo mismo que ya hicimos en el ejemplo del anterior capítulo.

El código en esta parte es muy parecido al del ejemplo anterior, excepto en que sustituimos el *gradient descent optimizer* por el *ADAM optimizer*, que es un algoritmo que implementa otro optimizador y que ofrece ventajas en este caso según la literatura del tema[41].

También incluimos el parámetro adicional *keep_prob*, que indica la probabilidad de *dropout* en el argumento *feed_dict* para controlar el proceso de *dropout* que hemos comentado antes.

```
cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

sess = tf.Session()

sess.run(tf.initialize_all_variables())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
     train_accuracy = sess.run( accuracy, feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.0})
     print("step %d, training accuracy %g"%(i, train_accuracy))
  sess.run(train_step,feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g"% sess.run(accuracy, feed_dict={ x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

Como en los casos anteriores, el código se puede encontrar en el *github* del libro y puede comprobarse que este código ofrece una exactitud de aproximadamente 99,2%.

Bien; aquí acaba esta introducción rápida a programar una *deep neural network*, entrenarla y evaluarla usando TensorFlow. Solo me queda comentar que el lector, si ha ejecutado el código, habrá notado que esta vez el entrenamiento de la red ha tardado bastante más que el anterior ejemplo de la red neuronal de una sola capa. ¿Se imaginan lo que podría llegar a tardar una red de muchas más capas? Le propongo que lea el siguiente capítulo para ver cómo podemos resolver esta situación si su servidor dispone de GPUs.

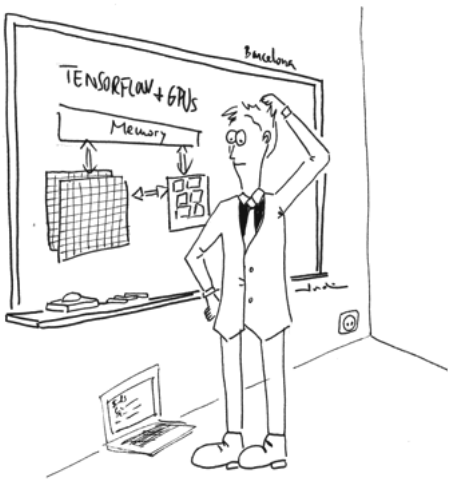
El lector puede encontrar el código que hemos usado en este capítulo en el fichero *CNN.py* del *github* del libro[42], aunque lo incluyo a continuación para facilitar su estudio con una visión global.

```
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
import tensorflow as tf

x = tf.placeholder("float", shape=[None, 784])
y_ = tf.placeholder("float", shape=[None, 10])

x_image = tf.reshape(x, [-1,28,28,1])
print "x_image="
print x_image

def weight_variable(shape):
```

```
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)

def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])

h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

sess = tf.Session()

sess.run(tf.initialize_all_variables())

for i in range(200):
   batch = mnist.train.next_batch(50)
   if i%10 == 0:
     train_accuracy = sess.run( accuracy, feed_dict={ x:batch[0], y_: batch[1], keep_prob: 1.0})
     print("step %d, training accuracy %g"%(i, train_accuracy))
   sess.run(train_step,feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g"% sess.run(accuracy, feed_dict={
       x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

[volver al índice de contenidos]

6. Uso de GPUs



Los servidores actuales pueden disponer de una o más GPUs. El paquete TensorFlow permite aprovechar este entorno para ejecutar la operación de entrenamiento simultáneamente en esas GPUs. En este breve capítulo presento un ejemplo introductorio de uso de las GPUs para aquellos lectores que quieran entender un poco más el funcionamiento de estos dispositivos de procesado.

Pero la versión actual del paquete TensorFlow permite usar las GPUs disponibles en un solo servidor. La versión distribuida que Google usa no está disponible de momento en el paquete TensorFlow.

Entorno de ejecución

La versión del paquete TensorFlow que soporta las GPU requiere el *Cuda Toolkit* 7.0 y CUDNN 6.5 V2. Para el proceso de instalación de todo el entorno, sugerimos que se visite la página *Cuda installation*[43] pues queda fuera del propósito de este libro ir más en detalle, y además la información en esta página estará actualizada.

La manera de referenciar estos dispositivos en TensorFlow es la siguiente:

- "/cpu:0": para referenciar la CPU del servidor.
- "/gpu:0": la GPU del servidor si solo hay una.
- "/gpu:1": la segunda GPU del servidor, y así sucesivamente.

Para saber a qué dispositivo se asignan nuestras operaciones y *tensores* hace falta crear la sesión con la opción de configuración *log_device_placement* instanciada a *True*. Veamos el siguiente ejemplo:

```
import tensorflow as tf
a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
c = tf.matmul(a, b)

sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
print sess.run(c)
```

Si el lector prueba este código en su ordenador, tendría que visualizar un *output* equivalente a:

```
. . .
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -&gt; device: 0, name: Tesla K40c, pci bus id: 0000:08:00.0
. . .
b: /job:localhost/replica:0/task:0/gpu:0
a: /job:localhost/replica:0/task:0/gpu:0
MatMul: /job:localhost/replica:0/task:0/gpu:0
. . .
[[ 22. 28.]
[ 49. 64.]]
. . .
```

Que, además del resultado de la operación, nos informa de dónde se ejecuta cada parte.

Si deseamos que una operación concreta se ejecute en un dispositivo determinado en lugar de que se esté seleccionado automáticamente por parte del sistema, podemos usar la variable *tf.device* para crear un contexto de dispositivo de manera que todas las operaciones dentro de ese contexto tengan la misma asignación de dispositivo.

Si tenemos más de una GPU en el sistema, la GPU con el identificador más bajo será seleccionada por defecto. Si desea ejecutar en una GPU diferente, tendrá que especificar la preferencia de forma

explícita. Por ejemplo, si queremos que el anterior código que se ejecute en la GPU 2 podemos usar el *tf.device('/gpu:2')*, como se indica en el siguiente código:

```
import tensorflow as tf

with tf.device('/gpu:2'):
a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
c = tf.matmul(a, b)
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
print sess.run(c)
```

Varias GPUs

Pero en realidad, si tenemos más de una GPU, lo que se quiere en general es usarlas todas a la vez para resolver un mismo problema. En este caso, podemos construir nuestro modelo de tal manera que podamos repartir el trabajo entre varias GPUs. Veamos el siguiente ejemplo:

```
import tensorflow as tf

c = []
for d in ['/gpu:2', '/gpu:3']:
with tf.device(d):
a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
c.append(tf.matmul(a, b))
with tf.device('/cpu:0'):
sum = tf.add_n(c)

sess = tf.Session()
print sess.run(sum)
```

Como podemos ver, es un código equivalente al anterior pero ahora tenemos dos GPUs indicadas por *tf.device* haciendo una multiplicación (las dos hacen lo mismo por simplificar el código en el ejemplo) y posteriormente el dispositivo CPU realiza la suma. Dado que hemos puesto el parámetro *log_device_placement* a *true*, se puede obtener por pantalla la siguiente información para ver cómo se han repartido los cálculos entre los diferentes dispositivos[44].

```
. . .
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -&gt; device: 0, name: Tesla K40c
/job:localhost/replica:0/task:0/gpu:1 -&gt; device: 1, name: Tesla K40c
/job:localhost/replica:0/task:0/gpu:2 -&gt; device: 2, name: Tesla K40c
/job:localhost/replica:0/task:0/gpu:3 -&gt; device: 3, name: Tesla K40c
. . .



. . .

Const_3: /job:localhost/replica:0/task:0/gpu:3
I tensorflow/core/common_runtime/simple_placer.cc:289] Const_3: /job:localhost/replica:0/task:0/gpu:3
Const_2: /job:localhost/replica:0/task:0/gpu:3
I tensorflow/core/common_runtime/simple_placer.cc:289] Const_2: /job:localhost/replica:0/task:0/gpu:3
MatMul_1: /job:localhost/replica:0/task:0/gpu:3
I tensorflow/core/common_runtime/simple_placer.cc:289] MatMul_1: /job:localhost/replica:0/task:0/gpu:3
Const_1: /job:localhost/replica:0/task:0/gpu:2
I tensorflow/core/common_runtime/simple_placer.cc:289] Const_1: /job:localhost/replica:0/task:0/gpu:2
Const: /job:localhost/replica:0/task:0/gpu:2
I tensorflow/core/common_runtime/simple_placer.cc:289] Const: /job:localhost/replica:0/task:0/gpu:2
MatMul: /job:localhost/replica:0/task:0/gpu:2
I tensorflow/core/common_runtime/simple_placer.cc:289] MatMul: /job:localhost/replica:0/task:0/gpu:2
AddN: /job:localhost/replica:0/task:0/cpu:0
I tensorflow/core/common_runtime/simple_placer.cc:289] AddN: /job:localhost/replica:0/task:0/cpu:0
[[ 44. 56.]
```

```
[ 98. 128.]]

. . .
```

Ejemplo de código

Para acabar este breve capítulo, presentamos un pequeño código inspirado en el que ha compartido Damien Aymeric en su *github*[45] que calcula A^n+B^n para *n=10* comparando el tiempo de ejecución con 1 sola GPU o con 2 GPUs usando el paquete *datetime* de Python.

Como siempre, empezamos importando los paquetes necesarios:

```
import numpy as np
import tensorflow as tf
import datetime
```

Creamos dos matrices de manera aleatoria con funciones del paquete *numpy* y el resto de código necesario para almacenar los resultados y especificar la operación matemática:

```
A = np.random.rand(1e4, 1e4).astype('float32')
B = np.random.rand(1e4, 1e4).astype('float32')

n = 10
```

Creamos dos estructuras de datos más para guardar los resultados:

```
c1 = []
c2 = []
```

A continuación definimos la función *matpow()* de la siguiente manera:

```
def matpow(M, n):
    if n &lt; 1: #Abstract cases where n &lt; 1
         return M
    else:
         return tf.matmul(M, matpow(M, n-1))
```

Como ya hemos visto, si queremos ejecutar el código con una sola GPU debemos especificarlo tal como se hace en el siguiente código:

```
with tf.device('/gpu:0'):
   a = tf.constant(A)
   b = tf.constant(B)
   c1.append(matpow(a, n))
   c1.append(matpow(b, n))

with tf.device('/cpu:0'):
   sum = tf.add_n(c1)

t1_1 = datetime.datetime.now()

with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
   sess.run(sum)

t2_1 = datetime.datetime.now()
```

Y para dos GPUs, se puede indicar de la siguiente manera:

```
with tf.device('/gpu:0'):
    #compute A^n and store result in c2
    a = tf.constant(A)
    c2.append(matpow(a, n))

with tf.device('/gpu:1'):
    #compute B^n and store result in c2
    b = tf.constant(B)
```

```
    c2.append(matpow(b, n))

with tf.device('/cpu:0'):
    sum = tf.add_n(c2) #Addition of all elements in c2, i.e. A^n + B^n

t1_2 = datetime.datetime.now()
with tf.Session(config=tf.ConfigProto(log_device_placement=True)) as sess:
    sess.run(sum)

t2_2 = datetime.datetime.now()
```

Finalmente, obtenemos los resultados con:

```
print "Single GPU computation time: " + str(t2_1-t1_1)
print "Multi GPU computation time: " + str(t2_2-t1_2)
```

El lector puede, por su cuenta, extender fácilmente el código a más GPUs si dispone de ellas. Como en los anteriores capítulos ell código aquí usado se encuetra en el *github* del libro. Espero que este capítulo haya sido suficientemente ilustrativo para ver cómo se puede acelerar el código con GPUs.

[volver al índice de contenidos]

Clausura

> *Exploration is the engine that drives innovation. Innovation drives economic growth. So let's all go exploring.*
> *Edith Widder*

Hasta aquí he presentado una introducción guiada de uso de TensorFlow para facilitar al lector el *warm-up* en esta tecnología que, sin duda, tendrá un papel destacado en el escenario tecnológico que se avecina. Como en realidad hay otras alternativas a TensorFlow que quizás se adaptan mejor a una necesidad concreta, invito al lector a explorar más allá del paquete TensorFlow.

Hay una gran diversidad en estos paquetes: algunos són más y otros menos especializados, algunos son más difíciles de instalar que otros, hay algunos muy bien documentados y otros que los que no es fácil encontrar información para saber con detalle cómo usarlo, a pesar que parece que funcionan muy bien, algunos permiten trabajar con GPUs y otros no…

Un dato: al día siguiente que Google liberaba TensorFlow, leía en un *tweet*[46] que durante el período 2010-2014 aparecía un nuevo paquete *Deep learning* cada 47 días, y en 2015 cada 22 días. Impresionante, ¿no creen? Como ya avanzaba en el primer capítulo del libro, una extensa lista como punto de partida para el lector la puede encontrar en la página *Awesome Deep Learning*[47].

Pero no solo hay alternativas a TensorFlow, también las hay a las redes neuronales como modelo, que a pesar de la gran popularidad de que estas gozan en estos momentos, las redes neuronales solo son una de las galaxias que conforman el inmenso universo *Machine Learning*.

Por poner un ejemplo, en vez de los miles de ejemplos que necesita para aprender una aproximación con un modelo *Deep Learning*, hay enfoques que tratan de aprender a partir de unos pocos ejemplos[48]; modelos que describen el proceso de generación de los datos y representan la incertidumbre de las distintas variables del modelo mediante la teoría de la probabilidad. Es lo que se conoce como *Probabilistic Machine Learning*[49].

Los métodos probabilísticos cubren la necesidad de hacer frente a la incertidumbre. La incertidumbre es inevitable en aplicaciones del mundo real, donde casi nunca podemos predecir con certeza lo que sucederá en el futuro, e incluso en el presente y el pasado, muchos aspectos importantes del mundo no se observan con certeza.

Pero, sea cual sea la aproximación a la analítica avanzada, estas son herramientas fundamentales y la clave de la nueva era de computación que se avecina. Supongo que el lector ya es consciente que nos encontramos en un nuevo punto de inflexión en la historia de la informática. A lo largo de su breve historia, la informática ha sufrido una serie de cambios profundos con diferentes etapas.

En su primera etapa, la informática consiguió hacer los números computables. Luego vino otro período, en el que nos encontramos, en que hizo el texto y la multimedia computable y sobretodo digitalmente accessible desde cualquier lugar y dispositivo.

Hoy en día, estamos experimentando ya la llegada de una nueva ola de cambios en el que se consigue hacer también computable el contexto, gracias a que los propios sistemas informáticos o simples dispositivos electrónicos incluyen capacidades predictivas, proveyendo la funcionalidad o contenidos correctos en el momento adecuado.

Estamos hablando de sistemas que ya estan aquí, interaccionando con los usuarios (a través de lenguaje natural o visualización) y que integran el conocimiento proveniente de cualquier fuente de información estructurada o no, que la contrastan con la experiencia pasada y con el estado actual de estos sistemas. Con toda esta información, estos sistemas pueden razonar para generar automáticamente hipótesis y valorar su efectividad.

Hay autores a los que les gusta enfatizar en que estos sistemas cogen funciones del celebro como: inferencia, predicción, correlación, abstracción, … dándole al sistema la posibilidad de hacer todo esto por si mismos. De aquí es donde sale la palabra "*cognitive*" para referenciar esta nueva computación: *cognitive computing* o computación cognitiva.

Pero todas estas capacidades de razonamiento requieren una nueva clase de sistemas de computación de altas prestaciones, así como de nuevos y potentes entornos *software* de ejecución que permitan hacer realidad esta analítica avanzada de manera más eficiente y automática para al usuario. Porque en realidad, e inexorablemente como pasa con la mayoría de las tecnologías digitales, el *Machine learning* está en proceso de ser automatizado y "comotizado" para acabar siendo transparente para la mayoría de sus usuarios.

Y esta es precisamente el área en la que estamos investigando desde el BSC y la UPC, trabajando en la integración de lo más avanzado en conocimiento de analítica, con los nuevos *frameworks* Big Data y con la impresionante potencia que presenta el *hardware* de los nuevos sistemas de computación de altas prestaciones. Esta área de investigación, *High-Performance Big-Data Analytics*[50], tiene por objetivo aportar soluciones que permitan avanzar en el campo de la computación cognitiva, sacando valor de los grandes volúmenes de datos disponibles en una gran variedad de formatos y generados a una velocidad sin precedentes hasta ahora.

Por ello, no me cabe la menor duda que cada vez más vamos a tener "ordenadores más sabios"[51] entre nosotros, con las múltiples implicaciones que esto representa para nuestra sociedad tal como la tenemos hoy en día concebida, y por ello está empezando a generarse un vivo debate acerca de qué consecuencias tendrá todo esto para nuestra sociedad. La historia nos muestra que cuando aparecen nuevas tecnologías disruptivas, estas tienen un efecto de polarización en la opinión pública, creando dos bandos: los optimistas y los pesimistas, los utopistas y los demasiado pragmáticos.

Recuerdo una entrevista[52] en *La Contra* de La Vanguardia al cirujano Marc Antoni Broggi, que decía que *"La medicina ha avanzado más en los últimos 25 años que en los últimos 25 siglos"*. Sin duda, una parte muy importante gracias a tecnologías como las que aquí hemos tratado. Pero también añadía *"¡hoy podemos hacer tantas cosas… que quizá sean demasiadas!"*. Al igual que el doctor Broggi, todo ingeniero e ingeniera tiene el deber de tener su propia opinión sobre las consecuencias del advenimiento de estas tecnologías que aquí hemos tratado y actuar en consecuencia de manera coherente con uno mismo.

Personalmente, pienso que la tecnología no es ni buena ni mala por si misma, sino que es simplemente una herramienta que se puede aplicar en innumerables situaciones útiles para las personas. Hay una infinidad de precedentes de cómo usamos de forma segura tecnología potencialmente peligrosa en la vida cotidiana, por ejemplo, los vehículos de motor o un simple cuchillo de cocina. ¿Se imaginan el presente sin muchas de las tecnologías que nos acompañan a diario? Mi opinión es que debemos avanzar mejorando la sociedad mediante el avance de la tecnología. Personalmente no creo que despertemos una mañana y nos encontremos a nuestro robot asistente doméstico intentando atacarnos con el cuchillo de cocina, ni a un robot jefe de las cadenas de producción de una fábrica, que por un exceso de celo decida eliminar todos los trabajadores humanos en aras de la eficiencia.

Pero está claro que el mercado laboral cambiará profundamente en los próximos años, despareciendo muchísimos de los perfiles de trabajo que hoy conocemos y apareciendo

muchísimos otros de nuevos, algunos que ni nos imaginamos en estos momentos. Este nuevo estadio representa un gran desafío a los trabajadores de cuello blanco de la sociedad del conocimiento, de la misma manera que la automatización de las fábricas en el siglo XX fue una revolución para los trabajadores de mono azul en las cadenas de montaje.

Así se refleja en el último informe titulado "*The Future of Jobs*"[53] del pasado *World Economic Forum* de Davos, en la que hablan de la cuarta revolución industrial, marcada por los avances en campos como la inteligencia artificial y el *machine learning*, la robótica, la nanotecnología, la impresión 3-D, la genética y la biotecnología.

Pero incluso en nuestro sector de las tecnologías de la información, uno de los más afortunados en los años venideros, en el que la demanda no parará de crecer y el número de ingenieros e ingenieras que se puedan formar dentro o fuera de las aulas será inferior a los que necesitan las empresas.

Es un efecto curioso el que se está produciendo. Como empleador de estos perfiles que también soy, veo que las empresas que somos sus contratadores ya nos estamos peleando para fichar a estos profesionales. Las entrevistas de contratación han cambiado en este sector, donde a menudo se trata de conseguir seducirlos contando las excelencias de nuestra empresa para que elijan fichar para nuestra empresa y no otra.

Pero me atrevo a augurar que esta misma tecnología de la que estamos hablando provocará que en un futuro no muy lejano se necesiten muchas menos manos para programar y menos cabezas para pensar algoritmos, arquitecturas de sistemas, etc. Habrá un punto de inflexión a partir del cual se precisen menos ingenieros e ingenieras, con la paradoja que los "culpables" de esta nueva situación seamos los mismos afectados que hacemos realidad estos nuevos sistemas de aprendizaje.

Esta tecnología mejorará nuestras vidas, pero con algoritmos que pueden predecir permitirá controlar lo que estamos apunto de hacer. La preocupación que mucha gente muestra por la privacidad pronto pasará a segundo plano cuando el reto sea salvaguardar la capacidad individual para decidir. Nos hace falta un debate social para prepararnos para la llegada de esta nueva era tecnológica que transformará profundamente la manera en que vivimos, trabajamos y pensamos.

[volver al índice de contenidos]

Agradecimientos

Escribir un libro requiere motivación pero también mucho tiempo, y por ello quiero empezar agradeciendo a mi familia el apoyo y la comprensión que ha mostrado ante el hecho de que un portátil compartiera con nosotros muchos fines de semana y parte de las vacaciones de Navidad desde que Google anunciara que liberaba TensorFlow el pasado noviembre.

A Oriol Vinyals le quiero agradecer muy sinceramente su disponibilidad y entusiasmo por escribir el prólogo de este libro, que ha sido para mí el primer gran reconocimiento al esfuerzo realizado. A Oriol lo conocí hace un par de años después de intercanviar unos cuantos correos electrónicos y en persona el año pasado. Realmente, un *crack* del tema de quien nuestro país se debería sentir muy orgulloso e intentar seducirlo para que algún día deje Silicon Valley y venga a Barcelona a fundar aquí nuestro propio Silicon Valley mediterráneo.

Como avanzaba en el prefacio del libro, un antiguo alumno licenciado en físicas e ingeniero en informática, además de haber sido uno de los mejores becarios de investigación que he tenido en el BSC, ha jugado un papel muy importante en esta obra. Se trata de Ferran Julià, que junto con Oriol Núñez han fundado una *startup*, con sede en mi comarca, en la que se preparan para analizar imágenes con redes neuronales convolucionales, entre otras muchísimas cosas que ofrece UNDERTILE. Este hecho ha permitido que Ferran Julià haya hecho a la perfección el rol de editor en este libro, incidiendo en forma y contenidos, de la misma manera que lo hizo mi editor Llorenç Rubió cuando publiqué con la editorial Libros de Cabecera mi ópera prima.

Ahora bien, a Oriol Núñez le agradezco profundamente la idea que compartió conmigo de ampliar las posibilidades de este libro y hacer llegar sus beneficios a muchas más personas de las que yo tenía en mente originalmente a través de su proyecto conjunto con la *Fundació El Maresme* para la

integración social y la mejora de la calidad de vida de las personas con discapacidad intelectual de mi comarca.

Mi más sincero agradecimiento a todos aquellos que han leído parcial o totalmente esta obra antes de ver la luz. En especial, a un importante *data scientist* como es Aleix Ruiz de Villa, quién me ha reportado interesantes comentarios para incluir en la versión que tienen en sus manos. Pero también a Mauro Gómez, Oriol Núñez, Bernat García y Manuel Carbonell por sus importantes aportaciones con sus lecturas previas.

Han sido muchos expertos en este tema que no conozco personalmente los que también me han ayudado en este libro, permitiéndome que compartiera sus ideas e incluso sus códigos, y por ello menciono en detalle las fuentes en los apartados correspondientes en este libro, más como muestra de agradecimiento que no para que el lector lo tenga que consultar.

Mi mayor agradecimiento a mi universidad, la Universitat Politècnica de Catalunya – UPC Barcelona Tech, que ha sido el entorno de trabajo que me ha permitido realizar mi investigación sobre estos temas y acumular los conocimientos que aquí quiero compartir. Universidad que además me ofrece dar clases en la Facultat d'Informàtica de Barcelona, a unos alumnos brillantes, quienes me animan a escribir obras como esta.

En la misma línea, quiero agradecer al centro de investigación *Barcelona Supercomputing Center* – Centro Nacional de Computación (BSC) y en especial a su director Mateo Valero, y los directores de *Computer Science* Jesús Labarta y Eduard Ayguadé, quienes me han permitido y apoyado siempre esta "*dèria*" que tengo de tener que estar "*parant l'orella*" a les tecnologías que vendrán.

Especialmente me gustaría mencionar a dos de mis colegas de la UPC, con quien estoy codo a codo iniciando esta rama de investigación más de "analítica": Rubèn Tous y Joan Capdevila han mostrado fe ciega en mis "*dèrias*" de exploración de nuevos temas para conseguir que nuestros conocimientos puedan aportar a esta nueva área llamada *High-Performance Big-Data Analytics*. Hacía tiempo que no disfrutaba tanto haciendo investigación.

Relacionado con ellos, agradecer a otro gran *data scientist*, Jesús Cerquides, del *Artificial Intelligence Research Institute* del CSIC , de quien a través de la codirección de una tesis doctoral estoy descubriendo una nueva y apasionante galaxia en el universo del *Machine Learning*.

Y no puedo olvidarme de quien aprendo muchísimo, estudiantes que su trabajo final de máster trata estos temas: Sana Imtiaz o Andrea Ferri.

Hablando de GPUs, gracias a Nacho Navarro, responsable del BSC/UPC *NVIDIA GPU Center of Excellence*, por facilitarme desde el primer momento el uso de sus recursos para "entrenar" a mis redes neuronales.

Mi agradecimiento al catedrático de la UPC Ricard Gavaldà, uno de los mejores *data scientist* con los que cuenta el país, que con mucha paciencia me llevo de la mano en mis inicios en este inhóspito para mi, pero apasionante, mundo del *Machine Learning* a mediados del 2006 creando junto con Toni Moreno-García, Josep Ll Berra y Nico Poggi el primer *team* híbrido de *Data Scientist* con *Computer Engineers*, ¡momentos inolvidables!

Gracias a esa experiencia nuestro grupo de investigación incorporó el *Machine Learning* con resultados tan brillantes como las tesis de Josep Ll. Berral, Javier Alonso o Nico Poggi, donde usábamos el *Machine Learning* en la gestión de recursos de los complejos sistemas de computación actuales. Desde entonces que me he quedado prendado del *Machine Learning*.

Pero no fue hasta unos años más tarde, en 2014, cuando con la incorporación al grupo de Jordi Nin y posteriormente de Jose A. Cordero, que no hice el paso adelante en "*Deep Learning*". Sin su estímulo por este tema hoy este libro no existiría. Gracias, Jordi y José, y espero que vuestros nuevos retos profesionales os reporten grandes éxitos.

Y no quiero olvidar a Màrius Mollà, quién me empujó a publicar mi primer libro *Empresas en la nube*. Desde entonces no para de insistir para cuándo la siguiente obra… ¡Pues aquí la tienes! Y a Mauro Cavaller, un gran colaborador de Màrius, cuya contribución fue clave en mi opera prima, me ha aportado esta vez una última revisión formal.

Gracias también a Laura Juan por su exquisita revisión del texto antes de que este viera la luz; sin su ayuda esta obra no tendría la calidad que tiene. A Katy Wallace, por poner nombre a esta colección

en la que se edita esta obra. Y a Bernat Torres, por haber creado la fantástica página web de este libro.

A Oriol Pedrera, un genio que domina todas las artes plásticas, que me ha acompañado con mucha paciencia en las diversas técnicas que he ido usando para realizar las ilustraciones del libro, que junto con Júlia Torres y Roser Bellido hemos ido concretando una y otra vez hasta encontrar la versión que encuentran en este libro. ¡Ha sido genial! En la parte artística no quisiera olvidarme del gran ebanista Rafa Jiménez quien acepto, sin rechistar, construirme a medida mi mesa de dibujo.

Agradecer al meetup *grup d'estudis de machine learning de Barcelona* por acoger la presentación oficial del libro, y a Oriol Pujol per aceptar impartir la conferencia que ha acompañado esta presentación del libro en el meetup.

Y también , muchas gracias a las entidades que me han ayudado a hacer difusión de la existencia de esta obra: la Facultad de Informática de Barcelona (FIB), la aceleradora de proyectos tecnológicos ITNIG, el Col·legi Oficial d'Enginyers Informàtics (COEINF), la Associació d'Antics Alumnes de la FIB (FIBAlumni), el portal de tecnología TECNONEWS, el portal iDigital y el Centre d'Excel·lència en Big Data a Barcelona (Big Data CoE de Barcelona).

Para acabar, una mención especial a la "penya cap als 50", la "colla dels informàtics" que después de 30 años todavía hacemos encuentros que dejan a uno cargadísimo de energía. El caso es que aquel fin de semana de noviembre que a la gente de Google desde Silicon Valley se le ocurrió sacar a la luz TensorFlow, yo lo pase con esta peña. Si yo no me hubiera cargado las pilas con ellos durante ese fin de semana, les aseguro que al día siguiente, cuando me planteé enfrascarme en escribir este libro no habría tenido la energía necesaria. Gracias Xavier, Lourdes, Carme, Joan, Humbert, Maica, Jordi, Eva, Paqui, Jordi, Lluís, Roser, Ricard y Anna. Ahora ya he agotado la energía, ¿cuándo volvemos a quedar?

[volver al índice de contenidos]

Acerca del autor

Jordi Torres es catedrático de la UPC Barcelona Tech y lidera temas de investigación en el Barcelona Supercomputer Center con una amplia experiencia de más de 25 años en actividades de investigación y docencia, con decenas de publicaciones científicas y participación en numerosos congresos científicos del área. Con una formación de ingeniero de computadores, su espíritu explorador y emprendedor le ha llevado a ser también un ingeniero de *big data* capaz de interactuar con los científicos de datos. Actualmente, su investigación se centra en la convergencia de la computación de altas prestaciones con el *big data* y su aplicación a los retos que plantea la analítica del *big data* o la computación cognitiva (lo que llamamos *High-Performance Big-Data Analytics*). Dada su extensa carrera profesional en diferentes roles, también realiza actividades de consultoría y estrategia relacionadas con las tecnologías de próxima generación y su impacto, y actúa como experto para varias organizaciones y empresas o como mentor de emprendedores. Una de sus pasiones es la divulgación científica, que lo ha llevado a escribir libros, dar decenas de conferencias y colaborar asiduamente con medios de comunicación generalista. Mantiene un blog sobre tecnología en *www.JordiTorres.eu*, a través del que comparte sus conocimientos y opiniones.

[volver al índice de contenidos]

Referencias

[1] The MNIST database of handwritten digits. [en línea]. Disponible en: http://yann.lecun.com/exdb/mnist [Consulta: 16/12/2015].

[2] *Github*, (2016) Códigos fuente de este libro [en línea]. Disponible en: https://github.com/jorditorresBCN/TutorialTensorFlow [Consulta: 16/12/2015].

[3] *Github*, (2016) Awasome Deep Learning. [en línea]. Disponible en: https://github.com/ChristosChristofidis/awesome-deep-learning [Consulta: 9/01/2016].

[4] Wikipedia, (2016). Neocognitron. [en línea]. Disponible en: http://en.wikipedia.org/wiki/Neocognitron [Consulta: 9/01/2016].

[5] Wikipedia, (2016). TOP500. [en línea]. Disponible en: https://en.wikipedia.org/wiki/TOP500 [Consulta: 19/12/2015].

[6] TOP500 List, (2016). Performance Development. [en línea]. Disponible en: http://www.top500.org/statistics/perfdevel/ [Consulta: 16/12/2015].

[7] Universia España, (2015) 15 cifras sorprendentes sobre el Big Data. [en línea]. Disponible en: http://noticias.universia.es/cultura/noticia/2015/10/01/1131820/15-cifras-sorprendentes-big-data.html [Consulta: 16/12/2015].

[8] Bernard Marr web page, (2015). [en línea]. Disponible en: http://www.forbes.com/sites/bernardmarr/

[9] Wikipedia, (2016). Tom M. Mitchell. [en línea]. Disponible en: https://en.wikipedia.org/wiki/Tom_M._Mitchell [Consulta: 19/12/2015].

[10] TensorFlow, (2016) GPU-related issues. [en línea]. Disponible en: https://www.tensorflow.org/versions/master/get_started/os_setup.html#gpu-related-issues [Consulta: 16/12/2015].

[11] TensorFlow, (2016) Download and Setup. [en línea]. Disponible en: https://www.tensorflow.org/versions/master/get_started/os_setup.html#download-and-setup [Consulta: 16/12/2015].

[12] TensorFlow: Large-scale machine learning on heterogeneous systems, (2015). [en línea]. Disponible en: http://download.tensorflow.org/paper/whitepaper2015.pdf [Consulta: 20/12/2015].

[13] TensorFlow, (2016) *Python API – Summary Operations*. [en línea]. Disponible en: https://www.tensorflow.org/versions/master/api_docs/python/train.html#summary-operations [Consulta: 03/01/2016].

[14] TensorFlow, (2016) TensorBoard: Graph Visualization. [en línea]. Disponible en: https://www.tensorflow.org/versions/master/how_tos/graph_viz/index.html [Consulta: 02/01/2016].

[15] Uno de los revisores de este libro ha indicado que además ha tenido que instalar el paquete python-gi-cairo.

[16] Wikipedia, (2016). Mean Square Error. [en línea]. Disponible en: https://en.wikipedia.org/wiki/Mean_squared_error [Consulta: 9/01/2016].

[17] Wikipedia, (2016). Gradient descent. [en línea]. Disponible en: https://en.wikipedia.org/wiki/Gradient_descent [Consulta: 9/01/2016].

[18] *Github*, (2016) Códigos fuente de este libro [en línea]. Disponible en: https://github.com/jorditorresBCN/TutorialTensorFlow [Consulta: 16/12/2015].

[19] TensorFlow, (2016) API de Python – Tensor Transformations [en línea]. Disponible en: https://www.tensorflow.org/versions/master/api_docs/python/ array_ops.html [Consulta: 16/12/2015].

[20] TensorFlow, (2016) Tutorial – Reading Data [en línea]. Disponible en: https://www.tensorflow.org/versions/master/how_tos/reading_data [Consulta: 16/12/2015].

[21] *Github*, (2016) Libro TensorFlow – Jordi Torres. [en línea]. Disponible en: https://github.com/jorditorresBCN/LibroTensorFlow/blob/master/input_data.py [Consulta: 9/01/2016].

[22] *Github*, (2016) Shawn Simister. [en línea]. Disponible en: https://gist.github.com/narphorium/d06b7ed234287e319f18 [Consulta: 9/01/2016].

[23] Wikipedia, (2016). Squared Euclidean distance. [en línea]. Disponible en: https://en.wikipedia.org/wiki/Euclidean_distance#Squared_Euclidean_distance [Consulta: 9/01/2016].

[24] TensorFlow, (2016) API en Python. [en línea]. Disponible en: https://www.tensorflow.org/versions/master/api_docs/index.html [Consulta: 16/12/2015].

[25] En realidad "_" es como cualquier otra variable, pero en Python muchos usuarios, por convención, lo usamos para descartar resultados.

[26] *Github*, (2016) Libro TensorFlow – Jordi Torres. [en línea]. Disponible en: https://github.com/jorditorresBCN/LibroTensorFlow [Consulta: 9/01/2016].

[27] TensorFlow, (2016) Tutorial MNIST beginners. [en línea]. Disponible en: https://www.tensorflow.org/versions/master/tutorials/mnist/beginners [Consulta: 16/12/2015].

[28] Neural Networks and Deep Learning. Michael Nielsen. [en línea]. Disponible en: http://neuralnetworksanddeeplearning.com/index.html [Consulta: 6/12/2015].

[29] The MNIST database of handwritten digits. [en línea]. Disponible en: http://yann.lecun.com/exdb/mnist [Consulta: 16/12/2015].

[30] Wikipedia, (2016). Antialiasing [en línea]. Disponible en: https://es.wikipedia.org/wiki/Antialiasing [Consulta: 9/01/2016].

[31] *Github*, (2016) Libro TensorFlow – Jordi Torres. [en línea]. Disponible en: https://github.com/jorditorresBCN/LibroTensorFlow/blob/master/input_data.py [Consulta: 9/01/2016].

[32] Google (2016) TensorFlow. [en línea]. Disponible en: https://tensorflow.googlesource.com [Consulta: 9/01/2016].

[33] Wikipedia, (2016). Sigmoid function [en línea]. Disponible en: https://en.wikipedia.org/wiki/Sigmoid_function [Consulta: 2/01/2016].

[34] Wikipedia, (2016). Softmax function [en línea]. Disponible en: https://en.wikipedia.org/wiki/Softmax_function [Consulta: 2/01/2016].

[35] TensorFlow, (2016) Tutorial MNIST beginners. [en línea]. Disponible en: https://www.tensorflow.org/versions/master/tutorials/mnist/beginners [Consulta: 16/12/2015].

[36] Neural Networks and Deep Learning. Michael Nielsen. [en línea]. Disponible en: http://neuralnetworksanddeeplearning.com/index.html [Consulta: 6/12/2015].

[37] *Github*, (2016) Libro TensorFlow – Jordi Torres. [en línea]. Disponible en: https://github.com/jorditorresBCN/LibroTensorFlow [Consulta: 9/01/2016].

[38] El lector puede entrar en más detalle sobre estos parámetros de las CNN a partir de los apuntes del curso CS231n – *Convolutional Neural Networks for Visual Recognition* (2015). [en línea]. Disponible en: http://cs231n.github.io/convolutional-networks [Consulta: 30/12/2015].

[39] GIMPS – *programa de manipulación de imágenes de GNU* Apartado 8.2. Matriz de convolución [en línea]. Disponible en: https://docs.gimp.org/es/plug-in-convmatrix.html [Consulta: 5/1/2016].

[40] TensorFlow, (2016*) Tutorials: Deep MNIST for experts*. [en línea]. Disponible en:https://www.tensorflow.org/versions/master/tutorials/mnist/pros/index.html [Consulta: 2/1/2016].

[41] TensorFlow, (2016) *Python API. ADAM Optimizer* [en línea]. Disponible en: https://www.tensorflow.org/versions/master/api_docs/python/train.html#AdamOptimizer [Consulta: 2/1/2016].

[42] *Github*, (2016) Códigos fuente de este libro [en línea]. Disponible en: https://github.com/jorditorresBCN/TutorialTensorFlow [Consulta: 29/12/2015].

[43] TensorFlow, (2016) GPU-related issues. [en línea]. Disponible en: https://www.tensorflow.org/versions/master/get_started/os_setup.html#gpu-related-issues [Consulta: 16/12/2015].

[44] Para poder entener el output debo aclarar que este código está ejecutado en un servidor con 4 Tesla K40 GPUs del Barcelona Supercomputing Center.

[45] *Github* (2016) Aymeric Damien. [en línea]. Disponible en: https://github.com/aymericdamien/TensorFlow-Examples . [Consulta: 9/1/2015].

[46] Twitter (11.noviembre.2015). Kyle McDonald: *2010-2014: new deep learning toolkit is released every 47 days. 2015: every 22 days.* [en línea]. Disponible en: https://twitter.com/kcimc/status/664217437840257024 [Consulta: 9/01/2016].

[47] *Github*, (2016) *Awesome Deep Learning*. [en línea]. Disponible en: https:// github.com/ChristosChristofidis/awesome-deep-learning [Consulta: 9/01/2016].

[48] Lake, Brenden M., Ruslan Salakhutdinov, and Joshua B. Tenenbaum. "Human-level concept learning through probabilistic program induction." *Science* 350.6266 (2015): 1332-1338. http://web.mit.edu/cocosci/Papers/Science-2015-Lake-1332-8.pdf

[49] Ghahramani, Z. (2015). "Probabilistic machine learning and artificial intelligence". Nature, 521(7553), 452-459.

[50] *¿Qué queremos decir con "High-Performance Big-Data Analytics?"* [en línea] Blog Jordi Torres Disponible en: http://jorditorres.org/whatwemeanby-high-performance-big-data-analytics [Consulta: 10/1/2016].

[51] La Vanguardia (2015) *Ordenadores más sabios* de M. Valero y J. Torres. [en línea] Disponible en: http://jorditorres.org/wp-content/uploads/2015/05/Screen-Shot-2015-05-19-at-16.33.33.png [Consulta: 10/1/2016].

[52] La Vanguardia (2015) Entrevista a J.A. Broggi en La Contra. [en línea] Disponible en: http://www.lavanguardia.com/lacontra/20111017/54231694065/vive-tu-vida-hasta-el-final-apropiate-de-tu-muerte.html [Consulta: 10/1/2016].

[53] *The future of jobs* [en línea] Disponible en: http://www.weforum.org/reports/the-future-of-jobs [Consulta: 18/1/2016].

[volver al índice de contenidos]

Acerca del libro

Conseguir el libro:

- Edición en papel a través del portal Amazon.es por 15 Euros + iva
- Edición en Kindle a través del portal Amazon.es por 5 Euros + iva
- Edición en digital en ebook (PDF) a través del portal Lulu.com por 5 Euros + iva

Fotos (link) de la presentación del libro el lunes 1 de febrero.

Colección WATCH THIS SPACE
Diseño de la colección: DesignedInBarcelona.com

Primera edición en lengua castellana: enero 2016 versión 1.0 impresa
Segunda edición en lengua castellana: febrero 2016 versión 2.0 Kindle
Tercera edición en lengua castellana: febrero 2016 versión 3.0 web
Cuarta edición en lengua castellana: febrero 2016 versión 4.0 ebook (pdf) ISBN 978-1-326-56702-6

Diseño de cubierta: Jordi Torres
Ilustraciones: Jordi Torres
Revisión ortotipográfica y de estilo: Laura Juan Merino
Editor: Ferran Julià Massó

Para citar el libro:
Hello World en TensorFlow, para iniciarse en la programación del Deep Learning
Jordi Torres, BSC & UPC Barcelona Tech, Barcelona, 2016
Ed. Undertile, ISBN 978-1-326-53238-3

[volver al índice de contenidos]
