Sistemas Cognitivos Artificiales

Redes neuronales en entornos *Big Data*

Índice

Esquema 3

Ideas clave 4

9.1. ¿Cómo estudiar este tema? 4

9.2. GPU para entrenamiento de redes neuronales profundas 4

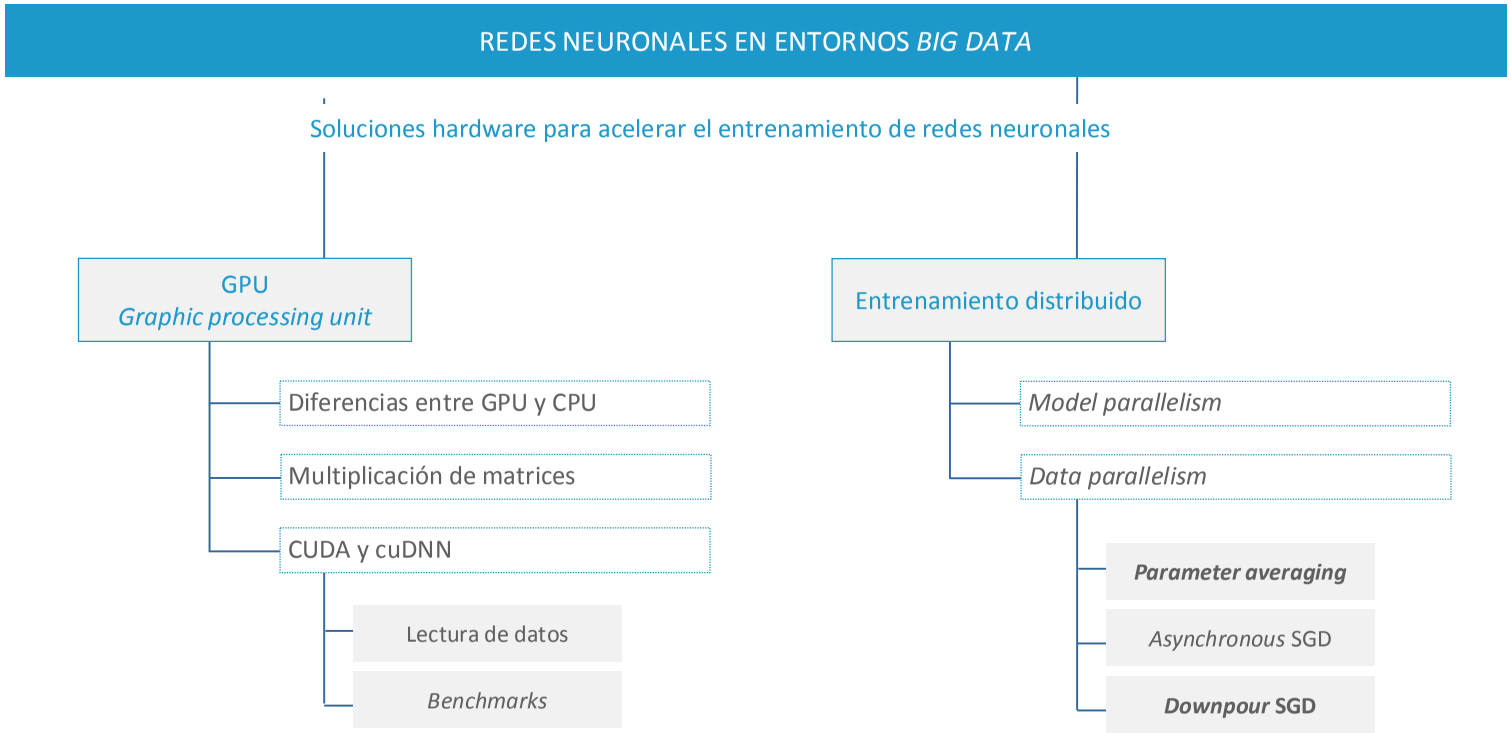
9.3. Entrenamiento distribuido 12

Lo + recomendado 21

+ Información 24

Test 26

Esquema



Ideas clave

9.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás leer las **Ideas clave** que se desarrollan a continuación.

En este tema veremos cómo acelerar el entrenamiento de redes neuronales con GPU y en sistemas distribuidos. Esto resulta fundamental para el entrenamiento de redes neuronales gigantes en entornos *Big Data*, donde la cantidad de datos disponibles es inmensa.

Es importante comprender en este tema por qué las GPU son capaces de acelerar el entrenamiento de los algoritmos de *deep learning*, así como las diferencias con las CPU que resultan claves para esto. Asimismo, es importante asimilar por qué factores como la velocidad de red o el ancho de banda de memoria influyen en la efectividad del uso de GPU y sistemas distribuidos. Otro punto esencial en este tema son los conceptos de *model parallelism* y *data parallelism*, así como entender qué ventajas y desventajas tienen algunas implementaciones particulares.

9.2. GPU para entrenamiento de redes neuronales profundas

Como comentamos al principio del curso, la gran disponibilidad de datos y la mejora de la capacidad de cómputo para entrenar redes neuronales han sido dos de los factores que han llevado al éxito del *deep learning*. En el aspecto de la capacidad de computación, la utilización de GPU para optimizar el proceso de entrenamiento ha jugado un papel clave. Esta **mejora en el proceso de**

entrenamiento ha permitido que la experimentación y la investigación con redes neuronales haya sido más rápida y efectiva, haciendo posible, además, la resolución de problemas casi intratables hasta entonces por la escala de tiempo necesaria. Por ejemplo, utilizando GPU, el tiempo que los investigadores necesitaban para entrenar la primera AlexNet era de alrededor de una semana.

Una tendencia clara en el mundo del *deep learning* a lo largo de los años es que los modelos se hacen más grandes y profundos, necesitando más operaciones para ser entrenados y, del mismo modo, aprovechando mejor los datasets mayores que van apareciendo. Por ello, alcanzar un entrenamiento rápido y eficiente de redes neuronales es un tema de gran importancia. En esta ocasión nos centraremos en los avances de hardware y de sistemas distribuidos para conseguir entrenar de manera eficiente en situaciones donde la cantidad de datos es muy grande, pudiéndose hablar de *Big Data*.

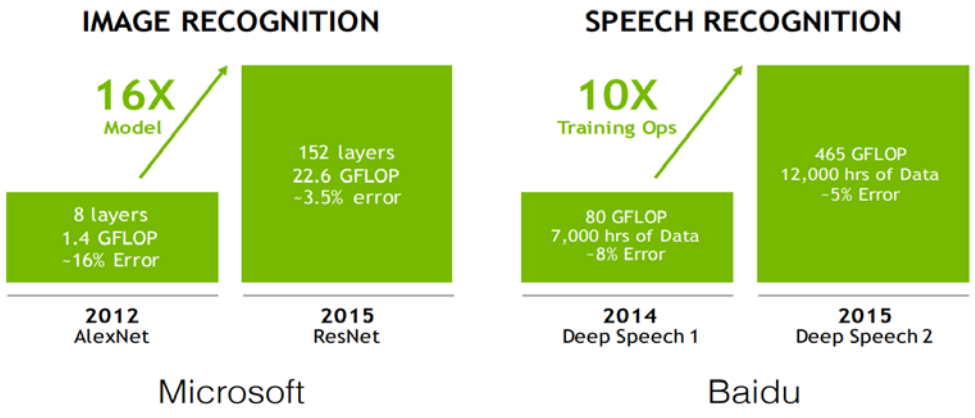


Figura 1. Ejemplo del aumento de modelos y operaciones para *Image* y *Speech Recognition*.
Fuente: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf

GPU: *graphics processing unit*

Una GPU *(graphics processing unit)* es un coprocesador hardware dedicado al procesamiento de gráficos y operaciones de coma flotante, con el objetivo de aligerar la carga de trabajo del procesador. Tradicionalmente, las GPU se han utilizado para videojuegos o aplicaciones gráficas con efectos avanzados, por ejemplo 3D. Visto así,

no es más que una tarjeta gráfica de toda la vida, si bien en el mundo del *deep learning* es preponderante la denominación GPU.

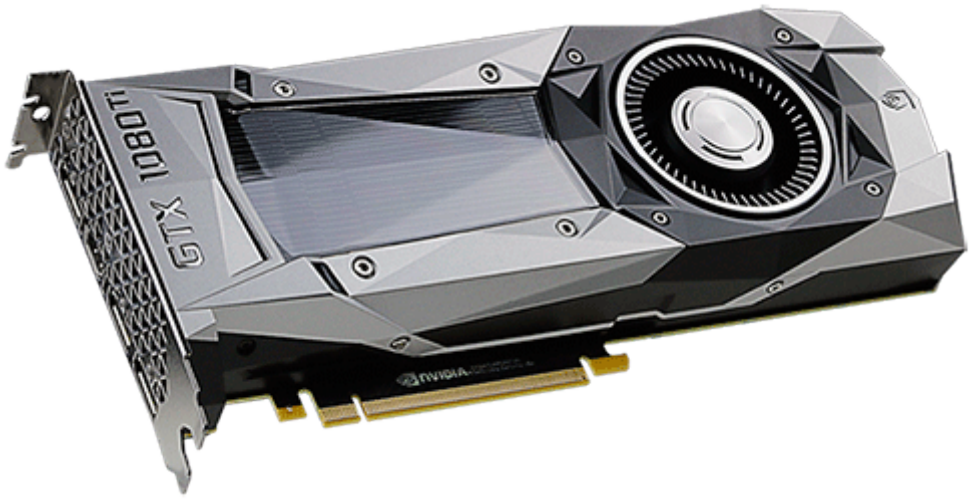


Figura 2. GTX 1080 Ti.
Fuente: https://www.evga.com/articles/01092/evga-geforce-gtx-1080-ti/

La labor de las GPU es, por tanto, complementar a la CPU en cierto tipo de operaciones para las cuales han sido diseñadas y optimizadas. Veamos **en qué se diferencian una CPU y una GPU**, tomando como ejemplo un procesador Intel i7-7700k y una NVIDIA GTX 1080 Ti.

Una primera diferencia es que las CPU suelen tener entre 2 y 10 *cores* o núcleos, esto es, son capaces de ejecutar en paralelo entre 2 y 10 procesos o hilos (en el ejemplo, 8 hilos con *hyperthreading*). La velocidad de reloj para estos puede llegar a ser de más de 4GHz.

Por otro lado, una tarjeta gráfica tiene varios miles de cores que corren, sin embargo, a una velocidad de reloj inferior.

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.2 GHz	System RAM	$339	~540 GFLOPs FP32
GPU (NVIDIA GTX 1080 Ti)	3584	1.6 GHz	11 GB GDDR5 X	$699	~11.4 TFLOPs FP32

Figura 3. CPU vs. GPU, hardware de 2017-2018.
Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf

- Como vemos en la diferencia de *cores*, una GPU puede ejecutar un número de tareas en paralelo mucho mayor que una CPU. Esta diferencia viene dada por el **tipo de trabajo** para el que cada cual ha sido diseñada:
  - Las CPU resuelven tareas de propósito general, mientras que las GPU están optimizadas para tareas muy particulares.
  - Las GPU tienen muchos *cores*, pero cada uno de estos es más lento y limitado a unas pocas operaciones, mientras que las CPU tienen pocos núcleos pero muy rápidos y capaces de realizar un gran número de tareas.

De este modo, las GPU destacan en su capacidad de paralelizar de manera masiva operaciones sencillas, mientras que las CPU están más orientadas a tareas secuenciales generales.

- Otra gran diferencia es el **uso de memoria**. Mientras que las CPU utilizan la memoria RAM del sistema, las GPU tienen su propia memoria RAM integrada.
  - La cantidad de memoria disponible en una GPU ha ido en aumento durante los últimos años, lo cual ha permitido entrenar modelos más grandes. El modelo (los parámetros de la red) tiene que estar en la memoria de la GPU, lo cual supone en muchas ocasiones una limitación. Esto lleva a sistemas complejos donde los modelos se dividen y entrenan entre varias GPU, como vimos en AlexNet.
  - Asimismo, las GPU están optimizadas para obtener grandes cantidades de memoria de manera rápida (tienen un mayor ancho de banda, *memory*

bandwidth). Mientras que las CPU son buenas en utilizar pequeñas cantidades de memoria de manera muy rápida, las GPU destacan en leer grandes cantidades de memoria de manera eficiente, si bien con mayor latencia.

Esto hace de nuevo que las tarjetas gráficas sean más efectivas para grandes operaciones numéricas como son las redes neuronales (recordemos que podemos tener millones de parámetros en una red).

Multiplicación de matrices en GPU

Veamos ahora, a modo de ejemplo, una **operación altamente paralelizable** que puede ejecutarse de manera muy eficiente en una tarjeta gráfica: la multiplicación de matrices.

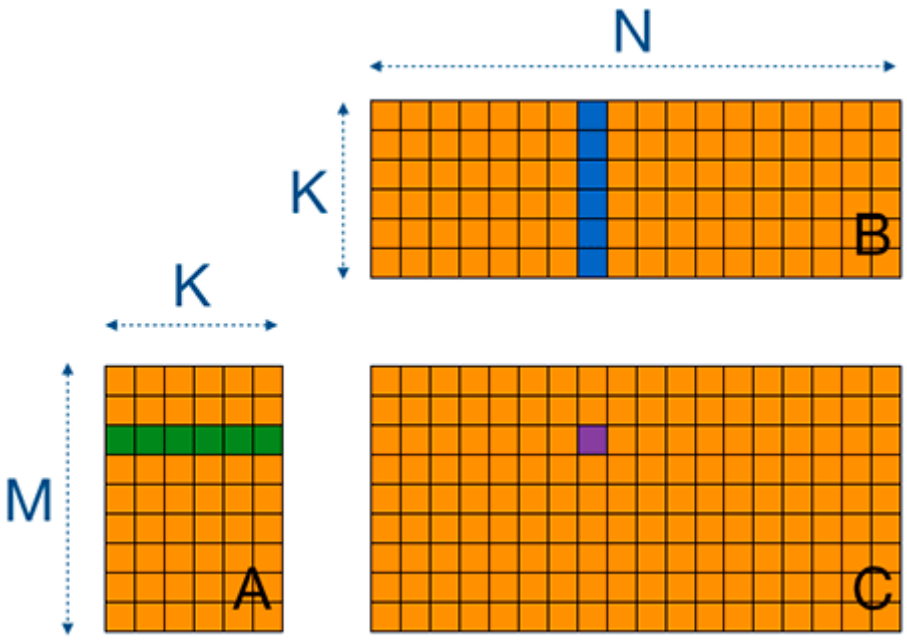


Figura 4. Multiplicación de matrices.
Fuente: https://cnugteren.github.io/tutorial/pages/page2.html

Como sabemos, cada elemento en la matriz resultante es el producto escalar de una fila y una columna de las matrices a multiplicar. Esto hace que podamos calcular cada

elemento de la matriz de salida de manera paralela, ya que no se necesita utilizar ningún valor intermedio.

Una GPU puede leer de manera rápida matrices gigantescas de una manera eficiente, pues tienen un gran ancho de banda de memoria, y utilizar sus miles de núcleos (otra gran diferencia respecto a las CPU) para **obtener de manera paralela todos los valores de salida**. Las operaciones a realizar son muy sencillas, ya que se trata simplemente de multiplicaciones y sumas de números reales, lo cual entra en el catálogo de operaciones relativamente simples que una GPU puede realizar.

A diferencia de la GPU, una CPU iría elemento a elemento de manera secuencial (o de 8 en 8, si dedicamos los 8 *cores* a 8 hilos distintos). El número de *cores* de una CPU se aleja bastante, como hemos visto, de los miles de cores disponibles en una GPU, por lo que es fácil ver de dónde viene la mejora de velocidad. Sin embargo, este ejemplo está en cierta modo muy simplificado, ya que las CPU actuales suelen tener operaciones vectorizadas y librerías algebraicas de alto rendimiento capaces de realizar multiplicaciones de matrices y otras operaciones numéricas de manera muy eficiente y utilizando varios procesadores.

Este ejemplo de la efectividad de las GPU calculando multiplicaciones de matrices es clave para entender por qué son tan útiles para el *deep learning*. Las redes neuronales, al fin y al cabo, son una serie de multiplicaciones de matrices. Todas las operaciones que hemos visto, capa a capa con los parámetros *w* de una red, pueden efectuarse en forma de producto de matrices. En muchos casos, productos de matrices enormes, ya que hay redes donde las capas pueden tener cientos o miles de elementos.

Del mismo modo, operaciones típicas como las convoluciones pueden ser también paralelizadas de manera sencilla. Todo esto permite que una GPU acelere el proceso de entrenamiento de una red neuronal en gran medida.

CUDA y cuDNN

De los dos fabricantes clásicos de tarjetas, NVIDIA y AMD, el primero se ha destacado en los últimos años por apostar fuerte en la utilización de sus GPU para *deep learning*, desarrollando incluso hardware optimizado para este fin. Una de las claves de su éxito ha sido CUDA, una plataforma de desarrollo para computación paralela con GPU. CUDA permite sacar el máximo provecho de las capacidades paralelas de las tarjetas gráficas, para lo que utiliza su propio lenguaje de programación, muy parecido a C.

Si bien desarrollar código CUDA no es sencillo, NVIDIA ofrece una serie de librerías y API que permiten sacar provecho de las capacidades de computación paralela de las GPU; una de estas es cuDNN, especialmente orientada a *deep learning*.

cuDNN implementa primitivas altamente optimizadas para el entrenamiento de redes neuronales profundas, tales como *forward* y *backward passes* de operaciones que hemos visto en este curso, como *dense layers*, *convolutions*, *pooling*, *batch normalization*, RNN, etc.

Librerías como cuDNN permiten a los desarrolladores centrarse en definir y entrenar modelos, facilitándoles abstraerse completamente de la complejidad de optimizar código para GPU. Casi todos los *frameworks* de *deep learning* que hemos visto en este curso, tales como TensorFlow y PyTorch, son capaces de acelerar sus operaciones con cuDNN, si esta librería está instalada y se dispone de una tarjeta gráfica NVIDIA. Aquí vemos de nuevo la ventaja que supone definir los *frameworks* en forma de un grafo de computación:

Las distintas operaciones del grafo pueden aprovechar una implementación más eficiente en GPU a partir de librerías de más bajo nivel como cuDNN.

CUDA y cuDNN solo funcionan en GPU de NVIDIA. Para tarjetas de ATI o para elementos de hardware más generales, existe una alternativa abierta llamada OpenCL.

Lectura de datos

La mejora de rendimiento que conlleva el uso de GPU puede provocar en ocasiones que el nuevo cuello de botella, en vez de ser las complejas operaciones matemáticas del entrenamiento de redes neuronales, sea la lectura y el procesamiento de los datos que utiliza la red para entrenar. El modelo está guardado en la RAM de la GPU, pero los datos suelen venir de ficheros en el disco duro.

Por ello, **para maximizar la cantidad de datos** que se puede alimentar a la red, conviene tener en cuenta ciertos factores:

- Si es posible, leer todos los datos a usar en la RAM de la máquina, de manera que la GPU tenga un acceso rápido a ellos.

- Utilizar un disco duro de estado sólido (SSD) en vez de un disco duro clásico (HDD). Los SSD son mucho más rápidos para lectura/escritura.

- Utilizar varios hilos de la CPU para leer y procesar datos. Esto es especialmente importante si necesitamos procesar el *training data* antes de pasarlo a nuestro algoritmo de aprendizaje. Por ejemplo, *tokenizar* el texto (separarlo en palabras), hacer transformaciones sobre imágenes, etc. Si utilizamos varios hilos de la CPU, podemos tener los datos preparados en un *buffer*, todo esto ocurriendo en paralelo al proceso de entrenamiento. El objetivo es conseguir que la GPU no se quede nunca esperando a que los datos estén listos.

Benchmarks

Es muy frecuente encontrar *benchmarks* donde se compara la velocidad de entrenamiento con distintos *frameworks* de *deep learning*, así como el uso de distintas librerías de optimización y tarjetas gráficas. Con estos *benchmarks*, es fácil ver cómo las GPU optimizan el proceso de entrenamiento de manera impresionante.

Como ejemplo, en la siguiente gráfica podemos ver los tiempos de entrenamiento para varias arquitecturas profundas utilizando solo CPU, GPU sin cuDNN y finalmente GPU con cuDNN.

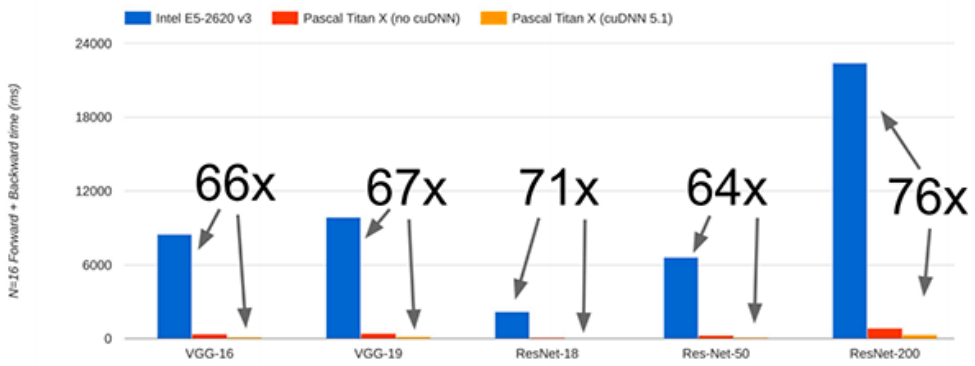


Figura 5. Tiempos de entrenamiento utilizando CPU, GPU sin cuDNN y con cuDNN.
Fuente: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf

9.3. Entrenamiento distribuido

En el apartado anterior, hemos visto cómo escalar el entrenamiento de redes neuronales con una gran cantidad datos en una sola máquina mediante la utilización de GPU. Sin embargo, ¿qué pasa si con esto no es suficiente? Una vez que ya no podemos hacer nuestro entrenamiento más rápido a base de tener una CPU y una o varias GPU más caras y potentes, no queda más remedio que empezar a buscar soluciones donde el entrenamiento se distribuya a través de varias máquinas.

Un sistema distribuido nos permite utilizar los recursos de muchas máquinas a la vez. Incluso en ocasiones puede ser más rentable utilizar varios servidores baratos en vez de solo uno con los últimos avances disponibles. Sin embargo, la utilización de sistemas distribuidos conlleva varias complejidades que hay que tener en cuenta:

- Los algoritmos de entrenamiento diseñados para un funcionamiento secuencial tienen que ser adaptados a un entorno donde varias máquinas hacen cálculos a la

vez. En nuestro caso, si distribuimos el cálculo de *stochastic gradient descent* a varias máquinas, este tiene que ser adaptado de alguna manera.

- Las máquinas del sistema distribuido han de tener cierta comunicación y coordinación entre ellas. Esto añade complejidad al sistema y nos obliga a tener en cuenta los tiempos de transferencia de datos por red, siempre más altos que el movimiento de datos en memoria RAM dentro de una sola máquina.

- El hecho de tener varias máquinas implica que la probabilidad de que una de ellas falle es más alta. Esto implica a su vez nuevas dificultades, ya que nos gustaría disponer de sistemas donde el entrenamiento no se detenga completamente si una máquina se cae.

Model parallelism y *data parallelism*

Para empezar, vamos a definir dos enfoques posibles para paralelizar o distribuir el entrenamiento de redes neuronales.

Model parallelism

En este enfoque, el modelo se distribuye en partes y cada máquina del sistema distribuido se hace responsable de los cálculos de una parte de la red, enviando los resultados correspondientes a otras máquinas. Por ejemplo, cada máquina podría realizar los cálculos de cada capa de una red.

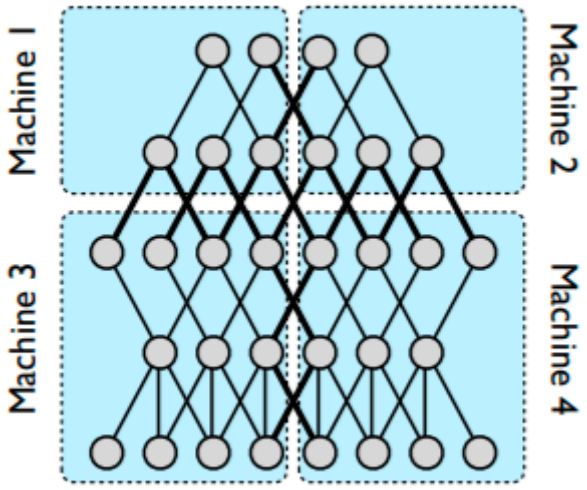


Figura 6. *Model parallelism*.
Fuente: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

En la imagen de arriba podemos ver cómo una red se parte en cuatro máquinas distintas. En general, cualquier grafo de computación puede partirse de manera similar. Las aristas en negrita denotan una transferencia de datos entre máquinas.

Data parallelism

Por otro lado, en *data parallelism* cada máquina tiene una copia completa del modelo y un subconjunto distinto de los datos de entrenamiento. Cada una va entrenando el modelo con sus datos y los distintos modelos resultantes son combinados cada cierto tiempo. En las siguientes secciones veremos ejemplos de cómo se hace esto.

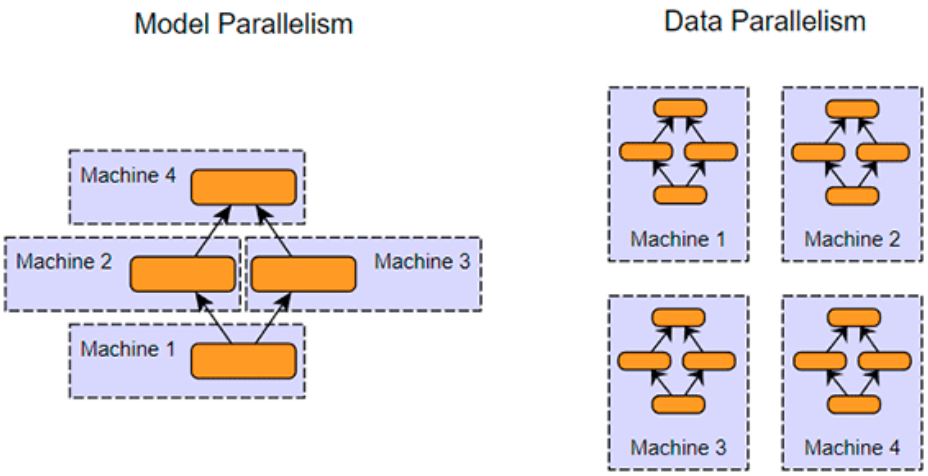


Figura 7. *Model parallelism* vs. *data parallelism*.
Fuente: https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-of-neural-networks/

Estos dos enfoques no se excluyen mutuamente. Por ejemplo, podríamos tener un clúster de máquinas en modo *data paralellism* donde cada máquina distribuye el modelo en varias GPU.

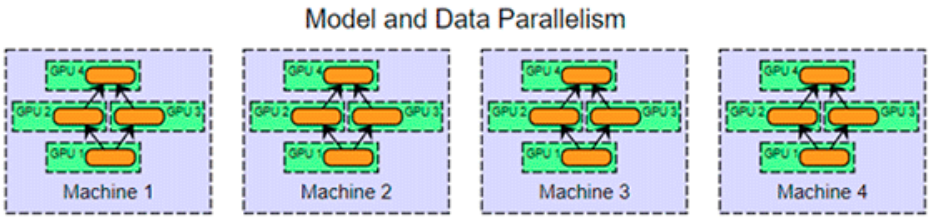


Figura 8. Híbrido de *Model parallelism* y *data parallelism*.
Fuente: https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-of-neural-networks/

Si bien *model parallelism* permite distribuir modelos que no caben en una sola máquina, el **esquema preferido** para el entrenamiento distribuido es *data parallelism*. Esto se debe a que es más sencillo de implementar, con una tolerancia a errores más fácil de tratar y con (normalmente) una utilización más efectiva de los recursos.

Parameter averaging

Es una de las formas más sencillas de aplicar *data parallelism*. En *parameter averaging* tenemos un servidor de parámetros (*parameter server*) encargado de mantener la versión actual del modelo a partir del trabajo de las distintas máquinas entrenando el modelo o *workers*. El proceso es el siguiente:

1. Los parámetros se inicializan de manera aleatoria siguiendo una de las estrategias vistas en clase.
2. El *parameter server* distribuye una copia de los parámetros actuales a cada *worker*.
3. Cada *worker* realiza el entrenamiento de una o varias *training batches* con su subconjunto de datos.
4. Cada *worker* envía los nuevos parámetros del modelo que ha obtenido al entrenar al *parameter server*.
5. El *parameter server* espera a recibir todos los parámetros de todos los *workers*. Una vez recibidos, se establecen los nuevos parámetros actuales del modelo como la media (*average*) de todos los parámetros recibidos.
6. Se vuelve al punto 2 y se repite el proceso durante el número de épocas (*epochs*) deseado.

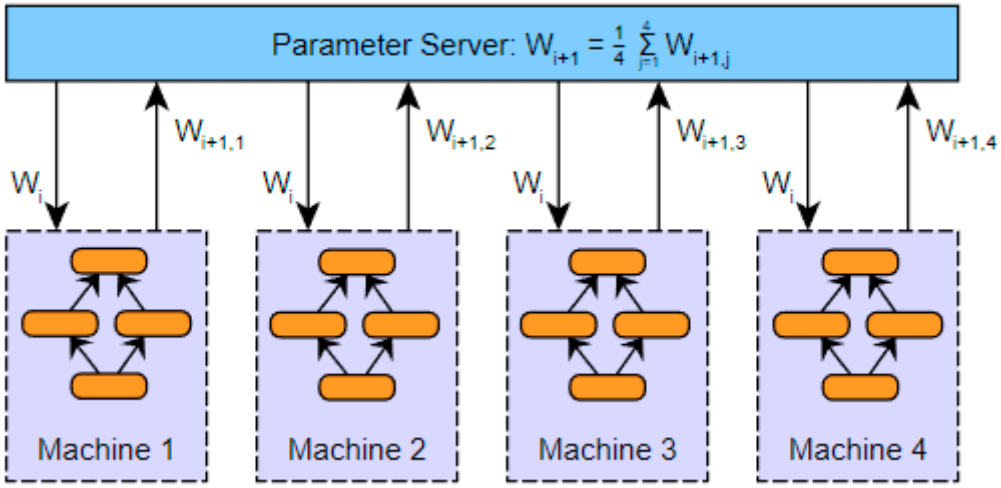


Figura 9. *Parameter averaging*.
Fuente: https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-of-neural-networks/

Uno de los aspectos clave a definir en este método es cada cuántas iteraciones se hace la media de los parámetros. Lo mejor sería hacerlo por cada iteración (*batch*), pero esto puede acabar haciendo que el coste de transferir los datos por la red al *parameter server* domine totalmente al coste de computación. Por otro lado, no se puede elegir un número muy grande de iteraciones, ya que cada máquina podría estar convergiendo a una solución distinta, con el peligro de que la media no dé lugar a un buen conjunto de parámetros.

Asimismo, *parameter averaging* tiene que esperar a que el trabajador (*worker*) más lento termine para hacer el *update* del modelo, lo que implica que acabamos perdiendo recursos mientras los otros trabajadores esperan. Del mismo modo, si uno de ellos falla, el proceso se detiene por completo, salvo que definamos alguna medida de protección como una media entre los trabajadores restantes.

Asynchronous SGD *y Downpour* SGD

***Asynchronous* SGD**

Esta es una versión para ejecución en paralelo de stochastic gradient descent. Como sabemos, SGD actualiza los parámetros de la red utilizando los gradientes de la función de coste a partir de *batches* de elementos de *training data* al azar.

Una forma de paralelizar SGD sería similar a la vista en *parameter averaging*, donde los distintos *workers* se sincronizan y van enviando sus *updates* (en este caso, los gradientes calculados en vez de los parámetros obtenidos). Esto, como hemos visto, produce ciertas ineficiencias en tanto que los *workers* tienen que esperar y sincronizarse entre ellos, en vez de actuar de manera independiente.

Una alternativa a esto es convertir el entrenamiento por SGD en paralelo en una especie de salvaje oeste donde cada *worker* calcula sus gradientes, los envía al servidor de parámetros y recoge de allí los nuevos parámetros calculados, probablemente conteniendo a su vez los *updates* de otros *workers*. La clave aquí es que el trabajador no tiene que esperar a que los otros terminen el proceso, sino que envía sus gradientes una vez ha terminado el cálculo y prosigue una vez recibe los parámetros actualizados.

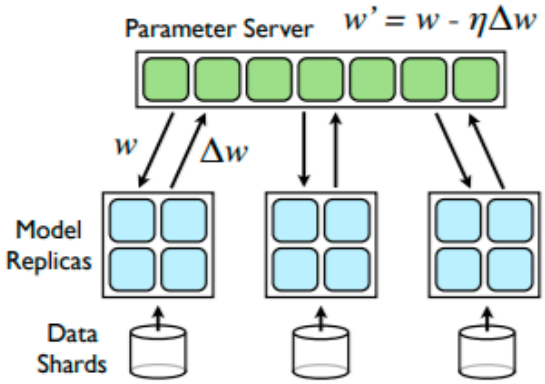


Figura 10. *Downpour* SGD.
Fuente: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

Downpour SGD

En *Downpour* SGD, otro sistema de distribución basado en *data parallelism*, el servidor de parámetros va recibiendo los gradientes de cada *worker* de manera asíncrona. Cuando los recibe, ejecuta la ecuación de SGD y obtiene los nuevos parámetros, que a su vez envía al *worker* que acaba de enviar los gradientes, ya que este necesita conocer cuál es el nuevo estado del modelo para seguir calculando. Como vemos, este proceso evita tener que esperar a los trabajadores lentos e, igualmente, no implica problemas si un trabajador falla.

Puede parecer extraño que un método así funcione. Al fin y al cabo, los *workers* están trabajando con parámetros del modelo que cambian continuamente y que, además, se quedan desactualizados cuando otros *workers* entregan sus *updates*. Este fenómeno, conocido como ***stale gradient problem***, puede dar lugar a problemas durante el entrenamiento. En la práctica, se aplican ciertas soluciones y el entrenamiento mediante *Asynchronous* SGD suele ser muy efectivo y converger de manera adecuada.

Downpour SGD fue implementado por Google en DistBelief, el precursor de TensorFlow, y en la actualidad se utiliza también en sistemas de producción. Algo que se hace de manera común cuando el número de *workers* es elevado es utilizar varios *parameter servers* en vez de uno.

Cada *parameter server* contiene parte de los parámetros del modelo para evitar que una sola máquina reciba los *updates* de todos los trabajadores. Como sabemos, los modelos de *deep learning* pueden llegar a tener millones o billones de parámetros, por lo que se puede producir un cuello de botella en la interfaz de red de la máquina encargada de recibir parámetros de todos los *workers*.

La distribución del *parameter server* en varias máquinas también puede crear problemas, ya que la caída de uno solo de ellos implica parar el entrenamiento hasta que el servidor vuelva. De manera similar, es importante distribuir los parámetros de

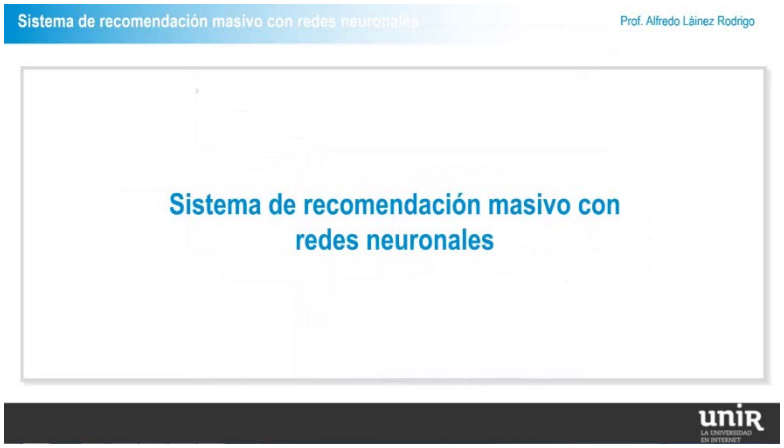
la red de manera adecuada entre los *parameter servers*. Por ejemplo, si estos servidores contienen *word embeddings* como parámetros y las palabras más comunes se guardan en el mismo servidor, este recibirá *updates* de manera mucho más frecuente, lo que puede colapsar su interfaz de red o crear un cuello de botella en la CPU al aplicar SGD para esos parámetros.

Lo + recomendado

Lecciones magistrales

Sistema de recomendación masivo con redes neuronales

Caso práctico de un sistema de *Deep learning* real en un entorno *big data*, concretamente el sistema de recomendación de YouTube basado en el artículo titulado *Deep Neural Networks for YouTube Recommendations*.



Accede a la lección magistral a través del aula virtual

No dejes de leer

Introducción al entrenamiento distribuido de redes neuronales

Skymind. (30 de noviembre de 2017). Distributed Deep Learning, Part 1: An Introduction to Distributed Training of Neural Networks [Blog post].

Blog post, que forma parte de una serie de tres partes, muy informativo sobre distintas estrategias de entrenamiento distribuido.

Accede al artículo a través del aula virtual o desde la siguiente dirección web: https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-of-neural-networks/

Large Scale Distributed Deep Networks

Dean, J. et al. (2012). Large Scale Distributed Deep Networks. *Proceedings of Neural Information Processing Systems 2012*.

Artículo de Google explicando el funcionamiento distribuido del framework de *deep learning* DistBelief.

Accede al artículo a través del aula virtual o desde la siguiente dirección web: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

CNN Benchmarks

Johnson, J. C. (2017). *CNN Benchmarks* [Archivo de datos y libro de códigos].

Benchmarks de arquitecturas CNN con GPU que hemos visto en este tema a la hora de comparar la velocidad de entrenamiento con distintos frameworks de *deep learning*, así como el uso de distintas librerías de optimización y tarjetas gráficas.

Accede al artículo a través del aula virtual o desde la siguiente dirección web:
http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

Which GPU to Get for Deep Learning

Dettmers, T. (21 de agosto de 2018). Which GPU(s) to Get for Deep Learning: My Experience and Advice for Using GPUs in Deep Learning [Blog post].

Post en el que se explica qué factores son importantes a la hora de elegir una GPU para *deep learning*, con análisis y recomendaciones. Nótese que el autor actualiza el contenido cada cierto tiempo.

Accede al artículo a través del aula virtual o desde la siguiente dirección web:
http://timdettmers.com/2018/08/21/which-gpu-for-deep-learning/

+ Información

A fondo

Efficient Methods and Hardware for Deep Learning

Clase magistral de Song Han en el curso CS231n de la Universidad de Stanford sobre algoritmos y hardware específico para acelerar el entrenamiento de redes neuronales.

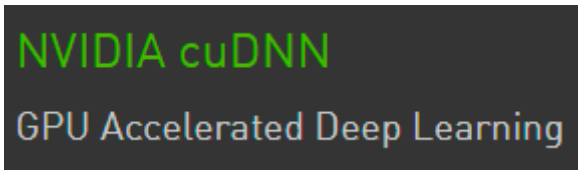
Accede al vídeo a través del aula virtual o desde la siguiente dirección web:

https://youtu.be/eZdOkDtYMoo

Webgrafía

cuDNN

Página web oficial de NVIDIA en la que podemos profundizar en cuDNN; en ella podemos encontrar desde enlaces a guías, foros hasta documentación de desarrollo.



Accede la página web a través del aula virtual o desde la siguiente dirección:

https://developer.nvidia.com/cudnn

Test

1. Las GPU (marca todas las respuestas correctas):

A. Están optimizadas para realizar operaciones gráficas, por eso son utilizadas solo con CNN.

B. Están optimizadas para realizar ciertas operaciones con números de coma flotante. Estas operaciones pueden ser utilizadas tanto para gráficos como para computación numérica.

C. Han permitido disminuir el tiempo de entrenamiento de las redes profundas en gran medida.

2. ¿Cuáles son las diferencias entre una CPU y una GPU? (Marca todas las respuestas correctas):

A. Las GPU tienen un número mucho mayor de *cores*.

B. Las GPU son un tipo especial de memoria inteligente, no tienen procesador.

C. Las GPU tienen memoria RAM propia, mientras que las CPU utilizan la memoria del sistema.

D. Las GPU utilizan la memoria RAM del sistema, mientras que las CPU tienen una memoria propia e independiente.

E. Las GPU están optimizadas para grandes anchos de banda de memoria, mientras que las CPU lo están para conseguir latencias bajas al leer de memoria.

3. ¿Qué elementos se necesitan guardar en la memoria de una GPU? (marca la respuesta correcta):

A. Todos los datos de entrenamiento.

B. Todos los datos de test.

C. Los parámetros del modelo.

4. En una multiplicación de matrices con GPU (marca la respuesta correcta):

A. La GPU lee las matrices a multiplicar de memoria y utiliza sus múltiples *cores* para obtener los valores resultantes de manera paralela, calculando en cada *core* un producto escalar.

B. La CPU y la GPU se coordinan, leen las matrices a multiplicar de memoria y suman sus *cores* para obtener los valores resultantes de manera paralela, calculando en cada *core* un producto escalar.

C. Ninguna de las anteriores.

5. cuDNN (marca todas las respuestas correctas):

A. Es una librería C genérica para operaciones en paralelo en GPU.

B. Define primitivas de operaciones comunes en *deep learning*.

C. Paraleliza de manera eficiente en código CUDA operaciones como convoluciones y RNN.

D. Es una librería de código abierto válido en todo tipo de arquitecturas.

E. Es utilizada por frameworks como TensorFlow, Caffe2 y MXnet.

6. En un sistema distribuido, la velocidad de la red de comunicaciones (marca la respuesta correcta):

A. Es importante solo para leer los datos de entrenamiento de manera rápida, que no suelen caber en una sola máquina y se encuentran a su vez distribuidos en un sistema tipo HDFS.

B. Es importante solo para mover los parámetros del modelo entre máquinas de manera rápida.

C. Es fundamental tanto para leer los datos de entrenamiento como para mover los parámetros del modelo entre máquinas.

D. No es tan importante como el conseguir el mayor número posible de máquinas.

7. Entrenar en un sistema distribuido (marca la respuesta correcta):

A. Debería hacerse siempre que sea posible para optimizar el entrenamiento.

B. Si utilizamos 20 máquinas, entrenaremos 20 veces más rápido.

C. Se ha de recurrir a ello de manera lógica cuando la escala de nuestros datos y modelos nos lleva a tiempos demasiados largos de entrenamiento para nuestra aplicación particular.

D. Ninguna de las anteriores.

8. El modelo CNN de AlexNet fue originalmente entrenado usando (marca la respuesta correcta):

A. *Data paralellism*.

B. *Model parallelism*.

C. Ambos, *data y model parallelism*.

D. Ningún tipo de paralelización.

9. En *parameter averaging* (marca todas las respuestas correctas):

A. No es necesario esperar a todos los *workers* para actualizar el modelo.

B. No es necesario tener un *parameter server*.

C. Cada *worker* tiene una copia completa del modelo.

D. Una vez se ha hecho la media de parámetros, cada *worker* recibe unos parámetros de vuelta distintos.

10. En *Downpour* SGD (marca todas las respuestas correctas):

A. No es necesario esperar a todos los *workers* para actualizar el modelo.

B. No es necesario tener un *parameter server*.

C. Cada *worker* tiene una copia completa del modelo.

D. Cada *worker* recibe unos parámetros de vuelta distintos al entregar sus *updates*.