Sistemas Cognitivos Artificiales

Roberto Casado Vara

Tema 9: Redes neuronales en entornos Big Data

UNIVERSIDAD INTERNACIONAL DE LA RIOJA

La Universidad en Internet

Redes neuronales en entornos Big Data

- GPUs para entrenamiento de redes neuronales profundas
- Entrenamiento distribuido

Tema 9.1: GPUs para entrenamiento de redes neuronales

GPUs para entrenamiento de redes neuronales

- 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*.

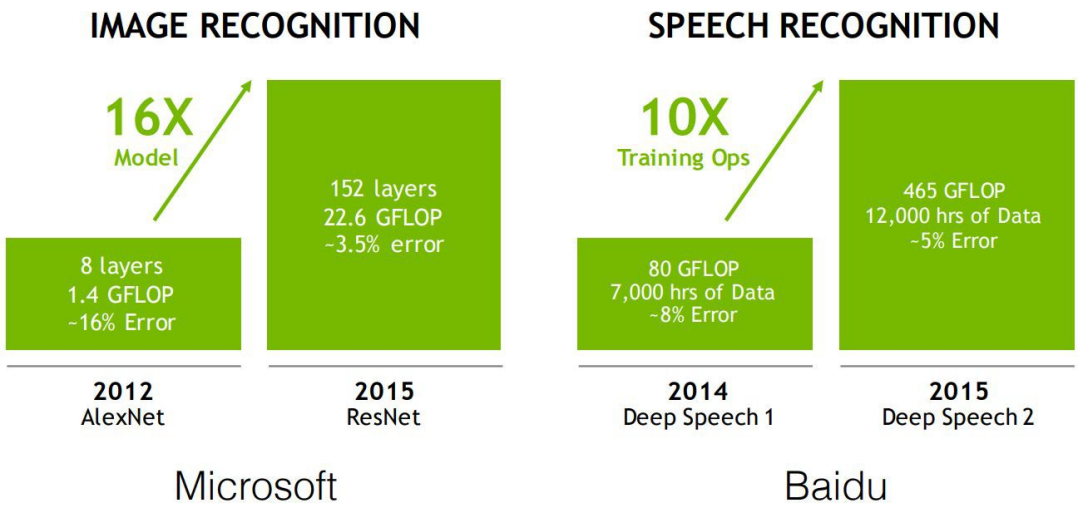
- La utilización de GPUs para optimizar el proceso de entrenamiento ha jugado un papel protagonista, permitiendo la utilización de más datos y de modelos mayores.

- Las GPUs han resultado claves para la resolución de problemas que eran intratables por la escala de tiempo necesario.
    - Tiempo original para entrenar *AlexNet* con GPUs: 1 semana

GPUs para entrenamiento de redes neuronales

- Tendencia en deep learning: modelos cada vez más grandes y profundos, necesitando más capacidad de cómputo para ser entrenados y aprovechando mejor datasets mayores.

- En este tema nos centraremos en los avances de *hardware* y sistemas distribuidos para conseguir entrenar de manera eficiente en entornos donde la cantidad de datos es enorme, pudiéndose hablar de Big Data.



Fuente de la imagen: Dally, NIPS 2016 workshop on Efficient Methods for Deep Neural Networks

GPU: Graphics Processing Unit

- Una **GPU** (***graphics processing unit*** o simplemente **tarjeta gráfica**) 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 utilizadas para videojuegos o aplicaciones gráficas con efectos avanzados, como 3D.

- Complementa a la CPU en cierto tipo de operaciones para las cuales las GPUs han sido diseñadas y optimizadas.

Diferencias entre GPU y CPU: número de *cores*

	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

- Las CPUs suelen tener entre 2 y 10 *cores* o núcleos a alta frecuencia, mientras que una GPU puede llegar a tener varios miles de cores a menor frecuencia.
  - A mayor número de núcleos, más capacidad de ejecutar procesos en paralelo.

- Una GPU puede ejecutar un número de tareas en paralelo mucho mayor. Sin embargo, estas tareas son mucho más simples.
  - CPU: tareas de propósito general, pocos *cores* pero muy potentes.
  - GPU: tareas muy particulares, muchos *cores* pero poco versátiles.

- Por tanto, las GPU destacan por su capacidad de paralelizar de manera masiva operaciones sencillas, mientras que las CPU están más orientadas a tareas secuenciales generales.

Fuente de la imagen: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf

Diferencias entre GPU y CPU: memoria

	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

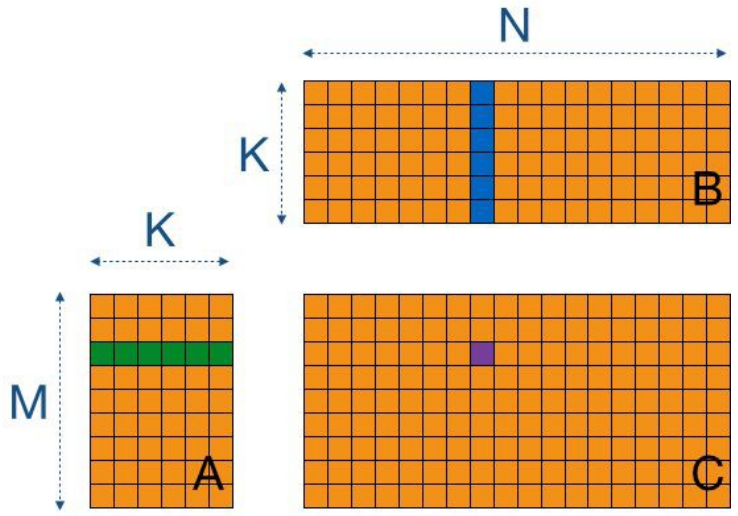
- La CPU usa la memoria RAM del sistema, mientras que una GPU tiene su propia memoria RAM integrada.
- La cantidad de memoria en una GPU ha ido en aumento en los últimos años. Esto permite entrenar modelos más grandes. En ocasiones, hay que recurrir a partir el modelo entre varias GPUs, como pasa con *AlexNet*.
- Las GPU están optimizadas para leer grandes cantidades de memoria de manera rápida (tienen un mayor ancho de banda), mientras que las CPU son buenas en utilizar pequeñas cantidades de memoria con menor latencia.
- Por tanto, las GPU son muy efectivas para operaciones numéricas con muchos datos.

Fuente de la imagen: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf

Multiplicación de matrices en GPU

- La multiplicación de matrices es un ejemplo de una operación altamente paralelizable que puede ejecutarse de manera muy eficiente en una GPU.

- Una GPU puede leer matrices de gran tamaño de manera eficiente (gracias al ancho de banda de su memoria) y puede utilizar sus miles de núcleos para, de manera paralela, obtener los valores de salida.



Fuente de la imagen: https://cnugteren.github.io/tutorial/pages/page2.html

Multiplicación de matrices en GPU

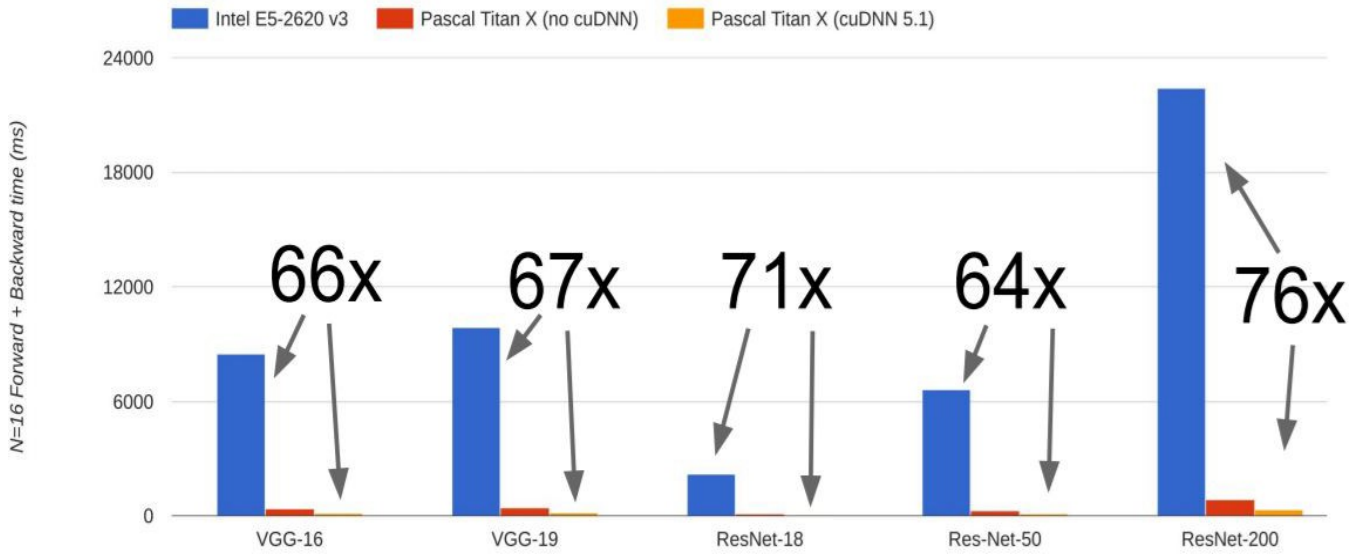
- La efectividad de calcular **multiplicaciones de matrices** con GPUs es una muestra de por qué estas aceleran el entrenamiento de redes neuronales. Como hemos visto, las redes neuronales funcionan principalmente a partir de multiplicaciones de matrices, que pueden llegar a ser muy grandes según el tamaño de las capas.

- Otras operaciones como las **convoluciones** también pueden ser paralelizadas de manera sencilla.

CUDA y cuDNN

- **CUDA** es una plataforma de desarrollo para computación paralela con GPUs. Utiliza su propio lenguaje de programación, parecido a C.
- **cuDNN** es una librería implementada con CUDA orientada a deep learning. Implementa primitivas altamente optimizadas (forward y backward passes) de fully connected layers, convolutions, pooling, batch normalization, RNNs, etc.
- Facilita a los desarrolladores abstraerse de la complejidad de optimizar código para GPU.
- Los *frameworks* de deep learning, como TensorFlow y PyTorch, son capaces de acelerar operaciones con cuDNN si se dispone de una GPU. **Las distintas operaciones del grafo de computación se pueden ejecutar con primitivas de cuDNN.**
- Solo disponible para tarjetas gráficas de Nvidia. Alternativa: OpenCL

CUDA y cuDNN



Fuente de la imagen: http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture08.pdf

Tema 9.2: Entrenamiento distribuido

Entrenamiento distribuido

- En ocasiones, la cantidad de datos a utilizar o el tamaño del modelo es tan grande que el uso de máquinas de gran potencia con GPUs no es suficiente.

- En estos casos se hace necesario distribuir el entrenamiento utilizando varias máquinas (sistema distribuido).

- En muchas ocasiones sale más rentable utilizar una red de máquinas baratas y de baja potencia que una sola de gran potencia. Sin embargo, los sistemas distribuidos conllevan varios problemas:

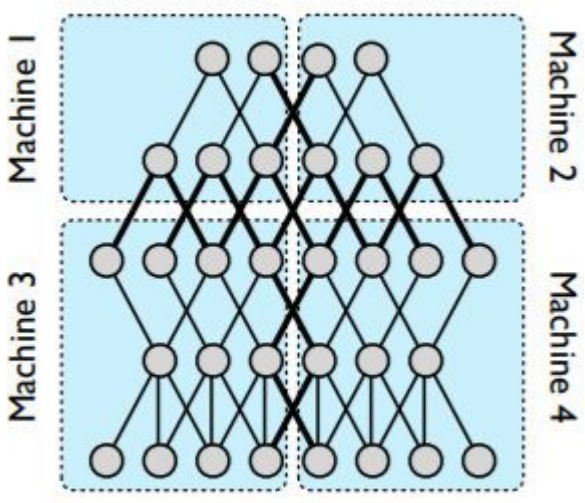
  - Los algoritmos de entrenamiento secuencial han de ser adaptados a un entorno donde varias máquinas hacen cálculos a la vez (en nuestro caso, SGD).
  - Las máquinas del sistema distribuido tienen que comunicarse entre sí, añadiendo complejidad y transferencia de datos por la red (más lento que la memoria RAM)
  - El hecho de tener varias máquinas aumenta la probabilidad de que una de ellas falle, pudiendo parar el entrenamiento.

- Vamos a ver dos formas de distribuir el entrenamiento: **model parallelism** y **data parallelism**

Model parallelism

- En **model parallelism**, el modelo se distribuye en partes y cada máquina del sistema distribuido se hace responsable de los cálculos de una parte del modelo, enviando los resultados correspondientes a otras máquinas.

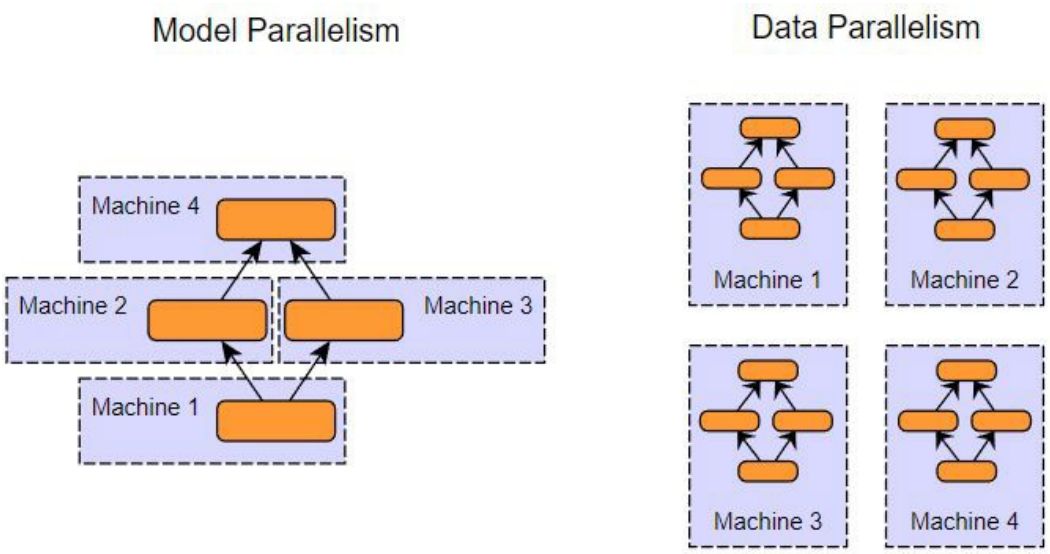
- Por ejemplo, cada máquina podría realizar los cálculos de una capa de una red. O, como vimos en *AlexNet*, las capas convolucionales más grandes pueden partirse.



Fuente de la imagen: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

Data parallelism

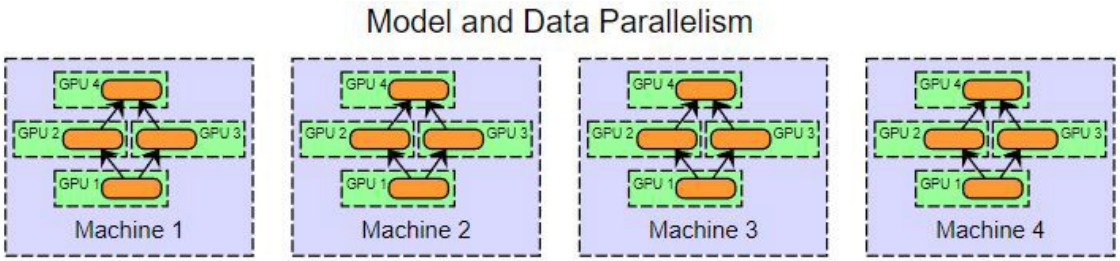
- En **data paralellism**, cada máquina tiene una copia completa del modelo y un subconjunto de los datos de entrenamiento. Cada máquina entrena el modelo con sus datos. Los distintos modelos resultantes son combinados cada cierto tiempo.



Fuente de la imagen: https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-ofneural-networks/

Entrenamiento distribuido

- *Model parallelism* y *data parallelism* no son mutuamente exclusivos. Podríamos tener un clúster de máquinas en modo *data parallelism* donde cada máquina distribuye el modelo en varias GPUs (*model parallelism*).



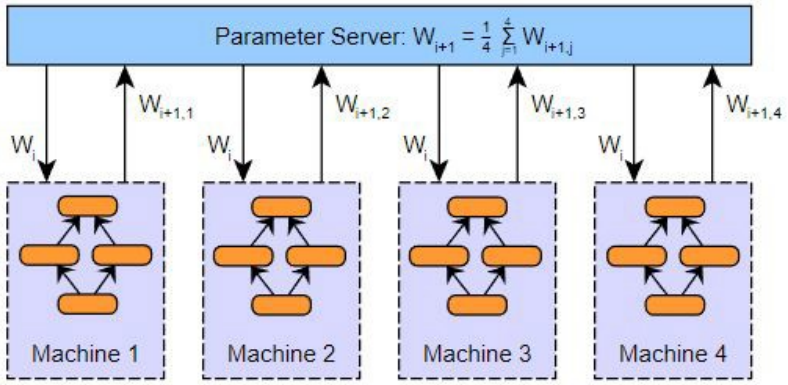
- Si bien *model parallelism* permite entrenar modelos que no caben en una sola máquina, en la práctica ***data parallelism*** **suele ser más utilizado**, ya que es más sencillo de implementar, con una mayor tolerancia a errores y (normalmente) una mejor utilización de los recursos.

Fuente de la imagen: https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-ofneural-networks/

Parameter averaging

- ***Parameter averaging*** es una forma sencilla de implementar un entrenamiento distribuido de tipo *data parallelism*.

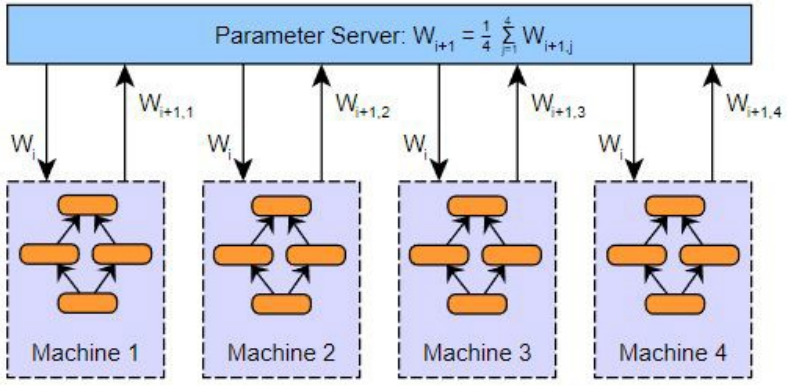
- En *parameter averaging*, tenemos un ***parameter server*** o servidor de parámetros, encargado de mantener la versión actual del modelo a partir del trabajo de las distintas máquinas entrenando el modelo, o ***workers***.



Fuente de la imagen: https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-ofneural-networks/

Parameter averaging

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 *epochs* deseado.



Fuente de la imagen: https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-ofneural-networks/

Asynchronous SGD / Downpour SGD

- *Parameter averaging* tiene el problema de que hay que esperar a que todos los *workers* terminen su iteración para actualizar los parámetros y hacer un *update* del modelo.

- Por tanto, el entrenamiento avanza a la velocidad del *worker* más lento.

- Igualmente, si un *worker* falla, el entrenamiento se detiene completamente.

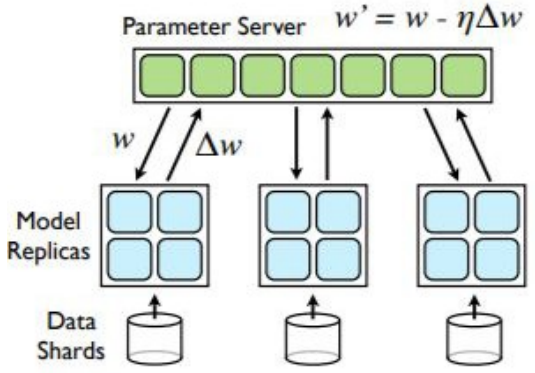
- ***Asynchronous SGD*** es una estrategia para paralelizar SGD al estilo del "salvaje oeste": cada *worker* calcula sus gradientes, los envía al servidor de parámetros y recoge de allí los nuevos parámetros calculados, probablemente conteniendo a su vez los updates de otros workers. La clave aquí es que **el *worker* no tiene que esperar a que los otros *workers* terminen el proceso**, sino que envía sus gradientes una vez ha terminado el cálculo y prosigue una vez recibe los parámetros actualizados.

Asynchronous SGD / Downpour SGD

- ***Downpour SGD*** es otra forma de entrenar una red neuronal de manera distribuida en base a *data parallelism*.

- Utiliza la idea de *asynchronous SGD.* El parameter server recibe los valores de los gradientes de cada *worker* de manera asíncrona y los utiliza para hacer un update inmediato de los parámetros guardados del modelo. Los nuevos parámetros son retornados al *worker.*

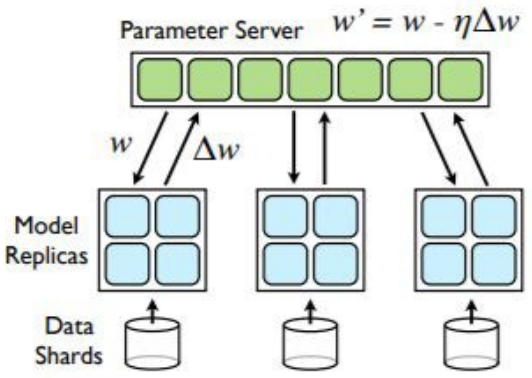
- Puede resultar raro que este sistema funcione (¡los parámetros que recibe un *worker* quedan desactualizados cuando otro *worker* envía sus cambios!), pero en la práctica los resultados son muy buenos.



Fuente de la imagen: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

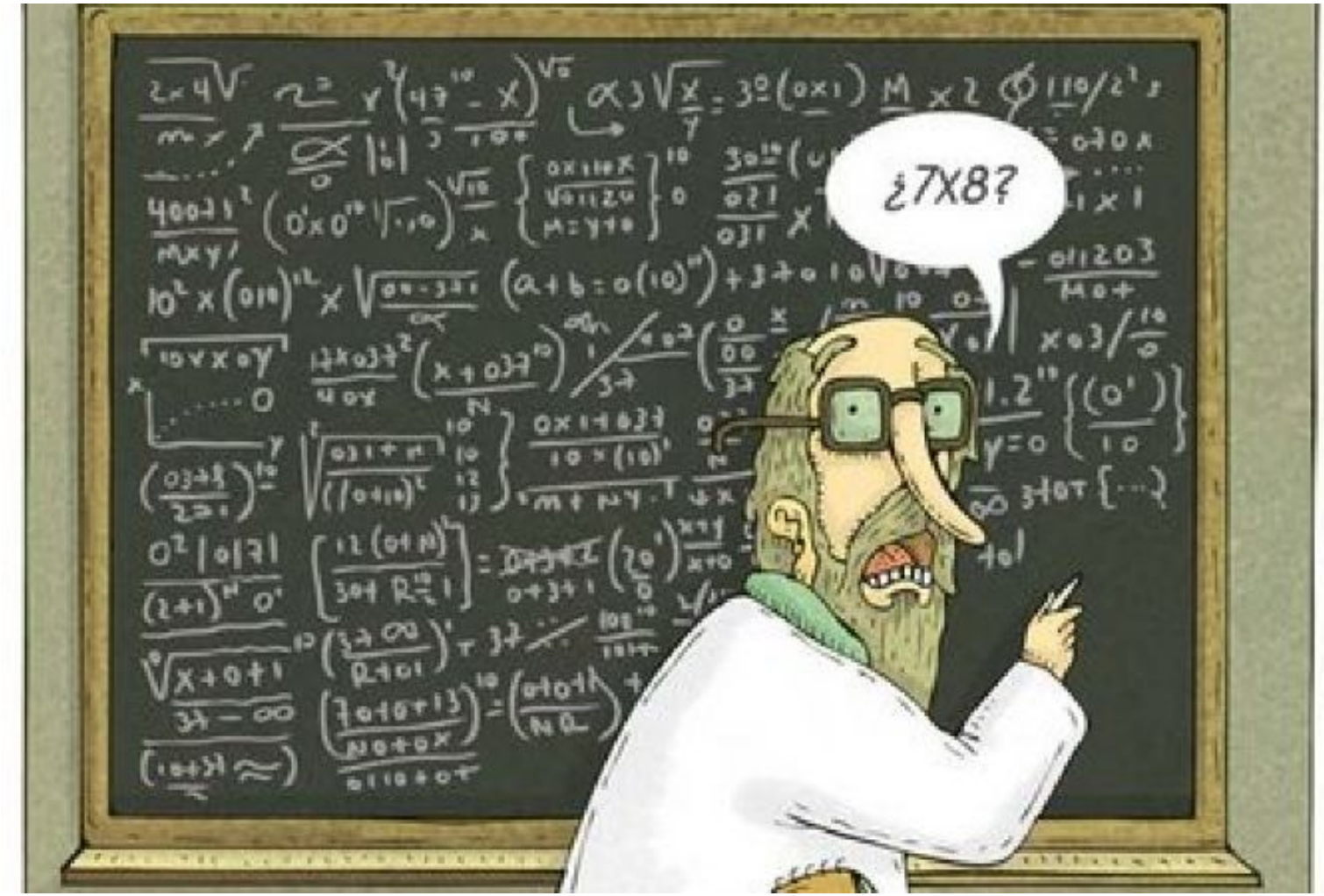
Asynchronous SGD / Downpour SGD

- ***Downpour SGD*** es comúnmente utilizado para entrenar modelos de forma distribuida.
- Cuando los modelos son enormes, se suelen distribuir los parámetros en varios *parameter servers*.
- Esto tiene ventajas (modelos mayores, las interfaces de red no se colapsan por la cantidad de datos) pero también problemas (si un *parameter server* falla, el entrenamiento se detiene, y el entrenamiento de nuevo depende del *parameter server* más lento).



Fuente de la imagen: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

¿Dudas?



www.unir.net