

Aprendizaje profundo (*Deep Learning*)

Aprendizaje profundo (en inglés, *deep learning*) es un conjunto de algoritmos de **aprendizaje automático** (en inglés, *machine learning*) que intenta modelar abstracciones de alto nivel en datos usando arquitecturas compuestas de transformaciones no lineales múltiples. (*Wikipedia*)

Vamos a las fuentes ➔ Yann LeCun, Yoshua Bengio & Geoffrey Hinton
Nature 521, 436–444 (28 May 2015)

REVIEW

doi:10.1038/nature14539

Deep learning

Yann LeCun^{1,2}, Yoshua Bengio³ & Geoffrey Hinton^{4,5}

Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics. Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and audio, whereas recurrent nets have shone light on sequential data such as text and speech.

Machine-learning technology powers many aspects of modern society: from web searches to content filtering on social networks to recommendations on e-commerce websites, and it is increasingly present in consumer products such as cameras and smartphones. Machine-learning systems are used to identify objects

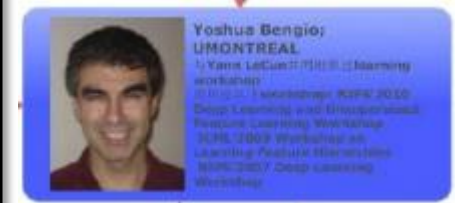
intricate structures in high-dimensional data and is therefore applicable to many domains of science, business and government. In addition to beating records in image recognition^{1–4} and speech recognition^{5–7}, it has beaten other machine-learning techniques at predicting the activity of potential drug molecules⁸, analysing particle accelerator data^{9,10}.



Geoffrey Hinton
University of Toronto



Yann LeCun
New York University



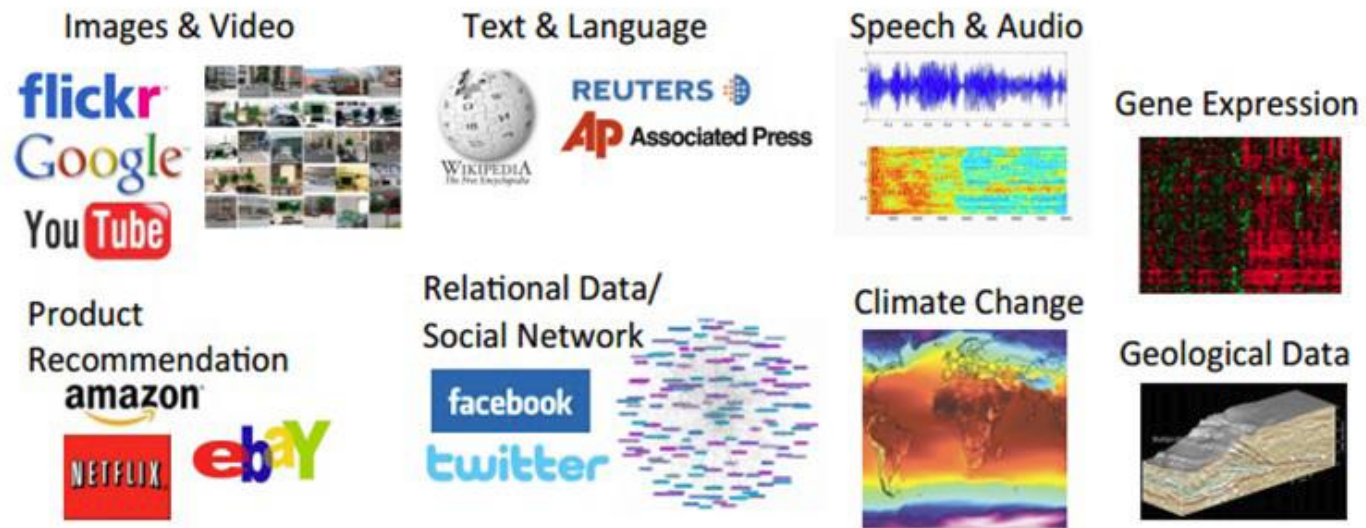
Yoshua Bengio
UNIVERSITÉ DE MONTRÉAL
1. Yann LeCun (1998) Learning
workshop
2. Yann LeCun (1998) Learning
workshop
3. Yann LeCun (1998) Learning
workshop
4. Yann LeCun (1998) Learning
workshop
5. Yann LeCun (1998) Learning
workshop
6. Yann LeCun (1998) Learning
workshop
7. Yann LeCun (1998) Learning
workshop
8. Yann LeCun (1998) Learning
workshop
9. Yann LeCun (1998) Learning
workshop
10. Yann LeCun (1998) Learning
workshop



Definición de DL:

“El aprendizaje profundo trata sobre el desarrollo de modelos computacionales compuestos de **múltiples capas** de procesamiento que aprenden **representaciones de datos** con múltiples niveles de abstracción. Su objetivo es descubrir una **estructura compleja** en **grandes** conjuntos de datos utilizando el algoritmo de propagación hacia atrás (*backpropagation*) para indicar cómo una máquina (en inglés *machine*) debe cambiar sus parámetros internos utilizados para calcular la **representación** en cada capa a partir de la representación en **la capa anterior**.”

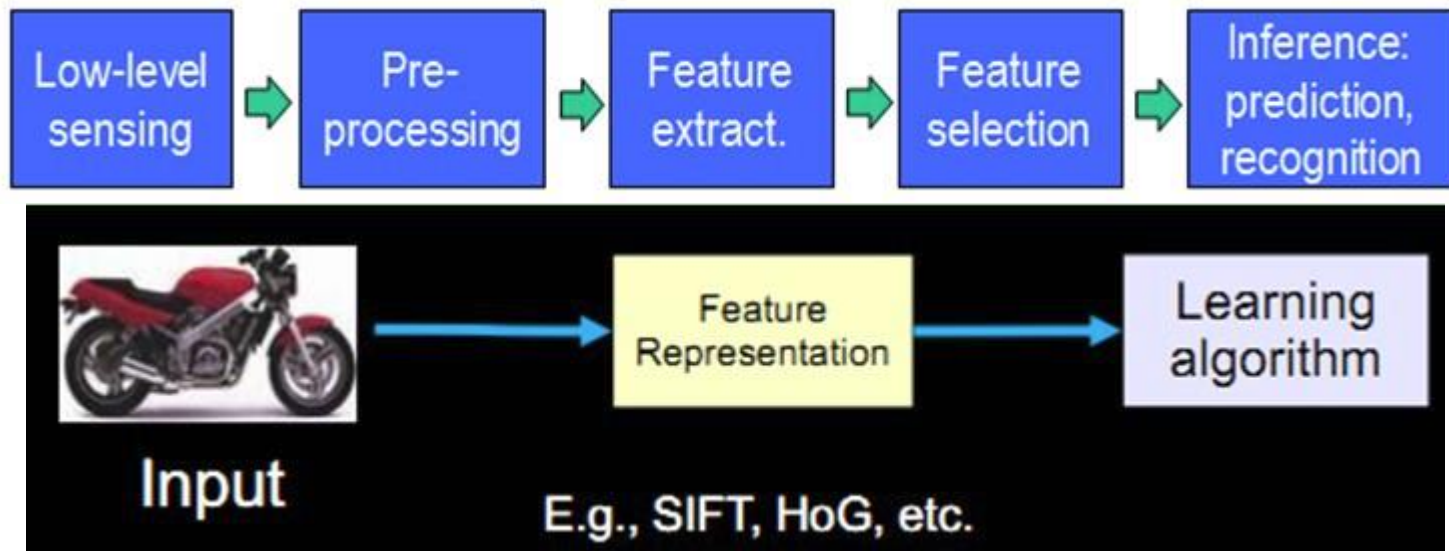
Estos métodos han mejorado dramáticamente el estado del arte en reconocimiento de voz, reconocimiento visual de objetos, detección de objetos y muchos otros dominios como el desarrollo de fármacos y en genética. *(del paper original de Deep Learning, LeCun, Bengio, Hinton)*



Comparación entre Aprendizaje Automático Clásico y Deep Learning

Aprendizaje Automático Clásico: Limitado en su capacidad para procesar datos en bruto (en especial los no estructurados).

Ingeniería de Atributos: requiere mucho tiempo y conocimiento del entorno por expertos.



SIFT=Scale-invariant feature transform, HOG; image histogram of oriented gradients.

Aprendizaje profundo:

Los métodos de aprendizaje profundo son métodos de aprendizaje de representación con múltiples niveles de representación.

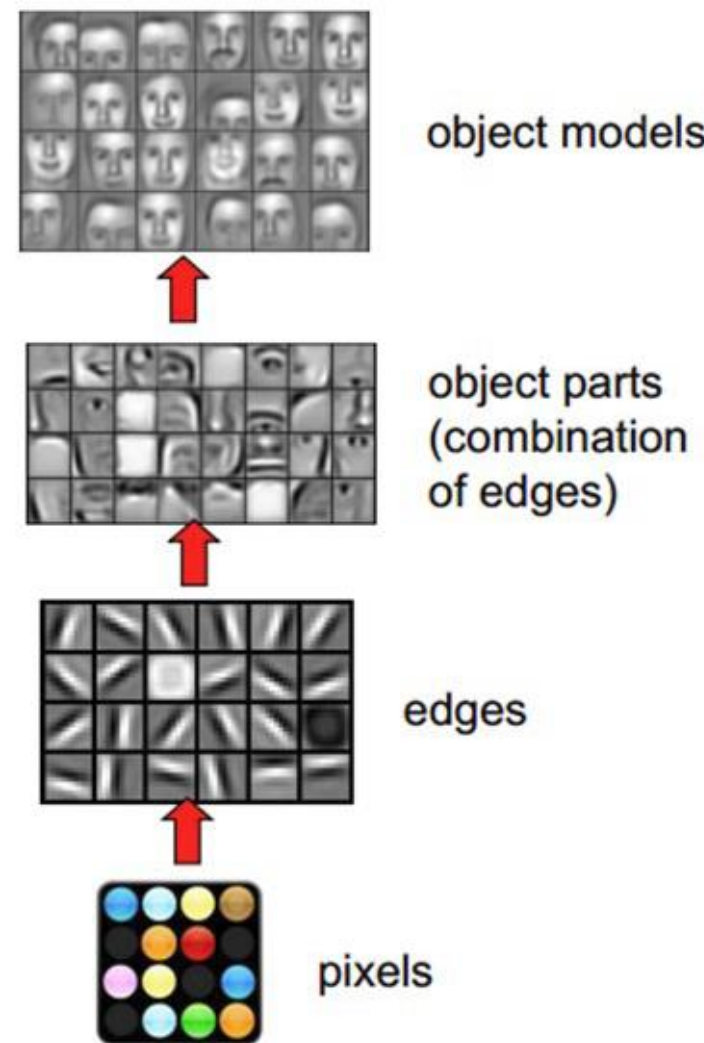
- Módulos simples pero no lineales de representación abstracta.
- mediante la composición de suficientes transformaciones de este tipo se pueden aprender funciones muy complejas.

Aprendizaje de representación:

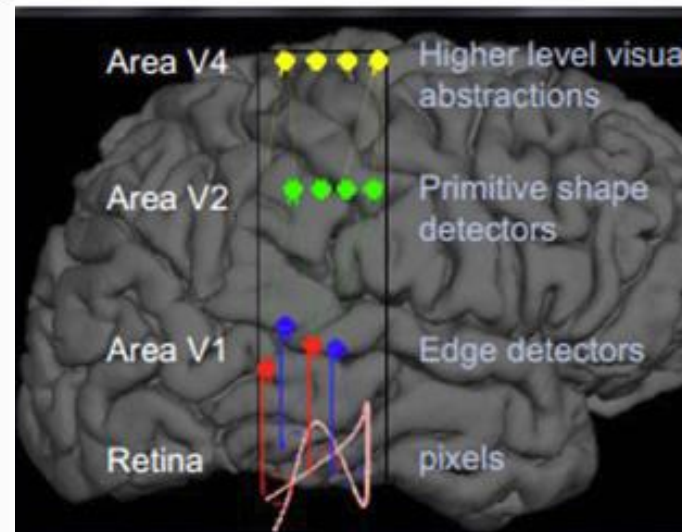
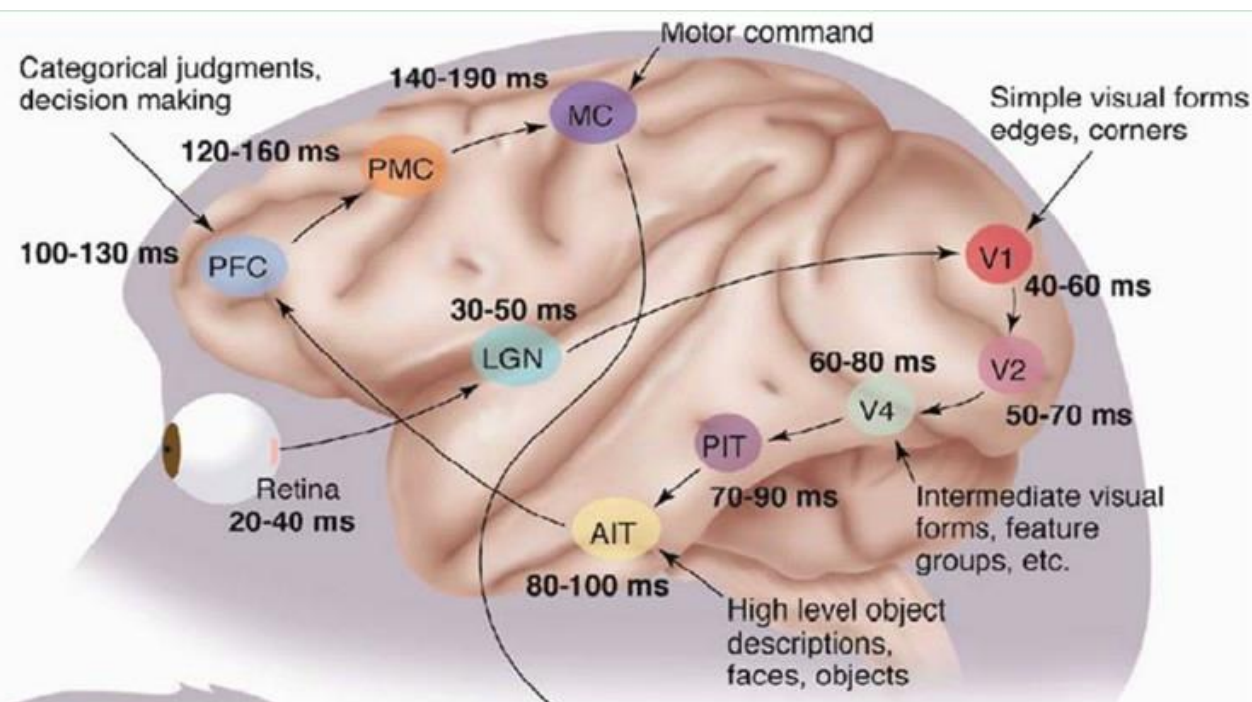
- Se alimenta con datos en bruto.
- Descubre automáticamente representaciones

Aspectos fundamentales:

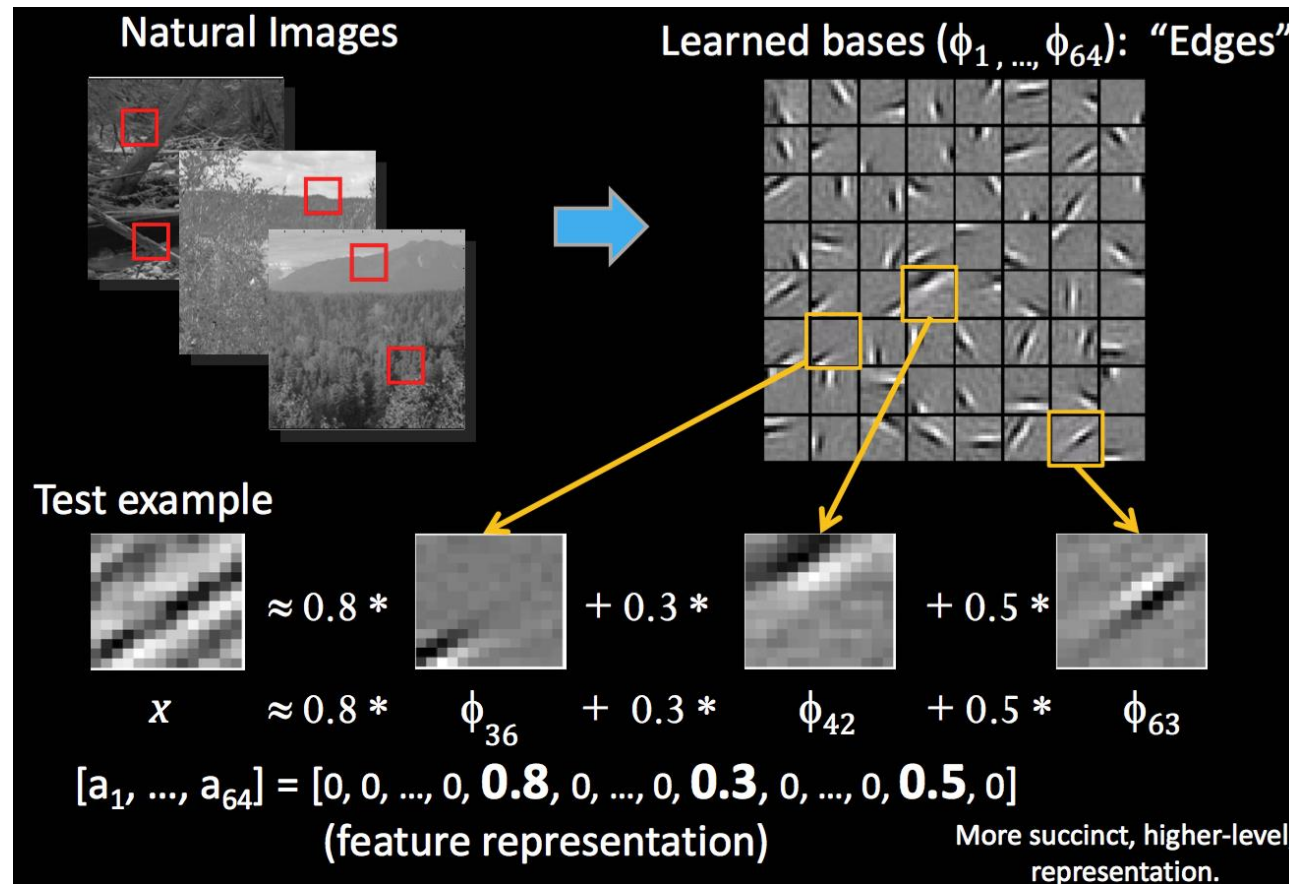
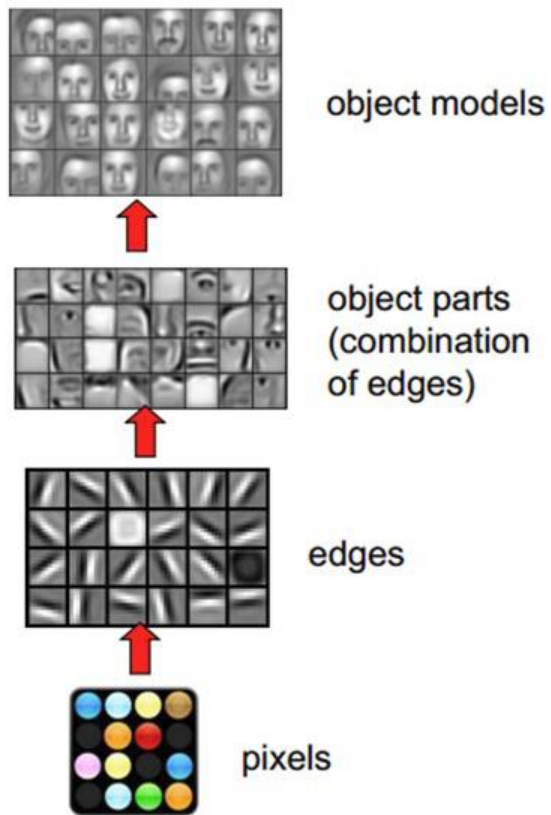
- Las capas de características no están diseñadas por humanos
- Aprende las características de los datos utilizando un procedimiento de aprendizaje genérico.



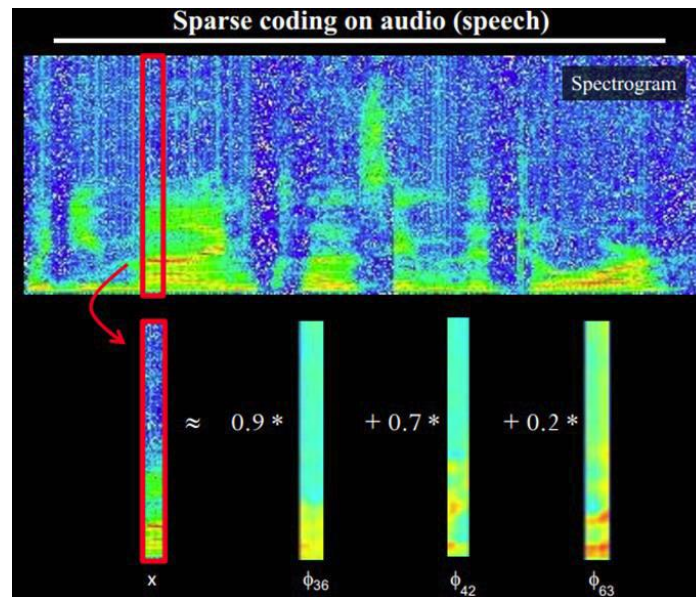
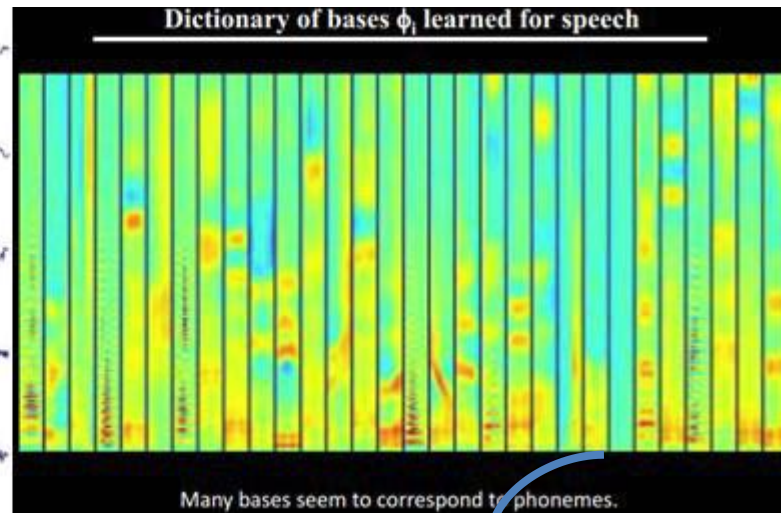
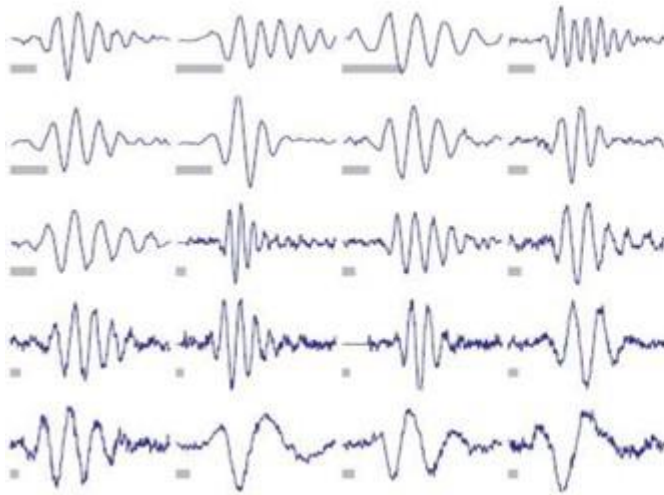
Córtex Visual y Red de Aprendizaje Profundo



Reconocimiento de Imágenes



Reconocimiento de Sonidos



Deep Learning en PC



El framework H2O permite a los usuarios de R el desarrollo de diferentes modelos de Machine Learning y Deep Learning que corren en un clúster H2O alojado en una máquina virtual de Java. Esta JVM y el clúster implementado explotan al máximo los recursos computacionales de la PC y permiten realizar cálculos en forma paralela.

Núcleos y threads

Product Specifications > Processors

Search specifications

Add to Compare

Intel® Core™ i7-3770 Processor

13 MB Cache, up to 3.90 GHz

of Cores

Cores is a hardware term that describes the number of independent central processing units in a single computing component (die or chip).

Processor	Code Name	Products formerly Ivy Bridge
Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz	Vertical Segment	Desktop

of Threads

A Thread, or thread of execution, is a software term for the basic ordered sequence of instructions that can be passed through or processed by a single CPU core.

Processor number	17-3770
Status	Discontinued

Performance

Especificaciones del dispositivo

Commodore	
Nombre del dispositivo	biblio01d
Procesador	Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz 3.90 GHz
RAM instalada	8,00 GB (7,90 GB usable)



H2O backend

¿Qué es H2O?

H2O es el motor de predicción en memoria de código abierto para Big Data Science. H2o permite a R usar todos los núcleos de la PC. R luego se ejecuta en múltiples núcleos de su computadora. LA PC es ahora como un clúster independiente.

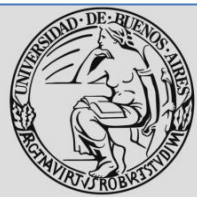
Explicación Técnica

H2O es un sistema de aprendizaje automático escalable distribuido. La arquitectura interna de H2O tiene un motor matemático distribuido (h2o-core) y una capa separada en la parte superior para algoritmos y GUI. La integración de *Mahout* requiere solo el motor matemático (h2o-core).

```
> h2o.init(nthreads = -1)
Connection successful!

R is connected to the H2O cluster:
H2O cluster uptime:      11 hours 43 minutes
H2O cluster timezone:    America/Argentina/Buenos_Aires
H2O data parsing timezone: UTC
H2O cluster version:     3.20.0.2
H2O cluster version age:  5 months and 15 days !!!
H2O cluster name:        H2O_started_from_R_charly_ghi948
H2O cluster total nodes: 1
H2O cluster total memory: 0.34 GB
H2O cluster total cores: 8
H2O cluster allowed cores: 8
H2O cluster healthy:     TRUE
H2O Connection ip:       localhost
H2O Connection port:     54321
H2O Connection proxy:    NA
H2O Internal Security:   FALSE
H2O API Extensions:      Algos, AutoML, Core V3, Core V4
```

H2O Cluster



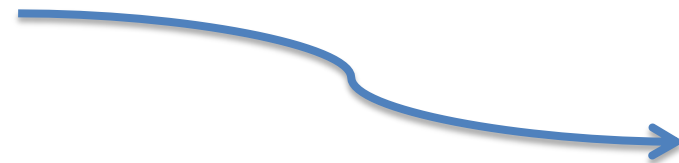


Qué es Mathout?

Apache Mahout (TM) es un framework de **álgebra lineal distribuida** y DSL matemático en Scala, diseñado para permitir que matemáticos, estadísticos y científicos de datos implementen rápidamente sus propios algoritmos. Tiene soporte para múltiples Backends distribuidos (incluyendo Apache Spark) y Soluciones nativas modulares para aceleración de CPU / GPU / CUDA

Mahout DRM

El módulo Matriz de fila distribuida DRM de Mahout (siglas en inglés de Distributed Row Matrix), es una abstracción para almacenar una gran matriz de números **en memoria** en un clúster mediante la **distribución de filas** entre los servidores. El DSL EN Scala de Mahout (llamado Samsara) proporciona una API abstracta de DRM para que los motores de back-end como Spark y H2O puedan realizar implementaciones mediante esta API.



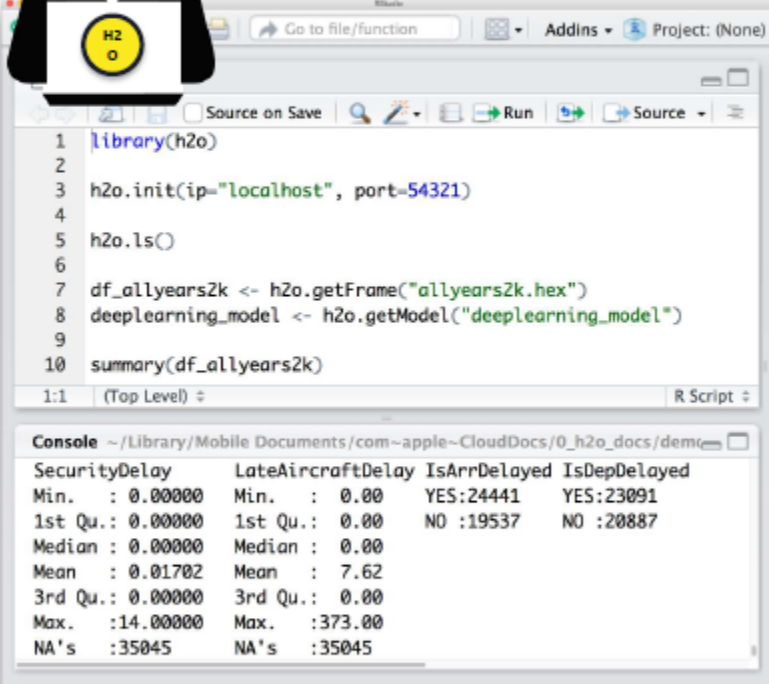

H2O Clients



H2O Cluster



H2O Clients




```
1 library(h2o)
2
3 h2o.init(ip="localhost", port=54321)
4
5 h2o.ls()
6
7 df_allyears2k <- h2o.getFrame("allyears2k.hex")
8 deeplearning_model <- h2o.getModel("deeplearning_model")
9
10 summary(df_allyears2k)
```

Console ~ /Library/Mobile Documents/com~apple~CloudDocs/0_h2o_docs/demc

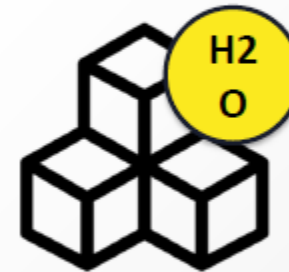
SecurityDelay	LateAircraftDelay	IsArrDelayed	IsDepDelayed
Min. : 0.00000	Min. : 0.00	YES:24441	YES:23091
1st Qu.: 0.00000	1st Qu.: 0.00	NO :19537	NO :20887
Median : 0.00000	Median : 0.00		
Mean : 0.01702	Mean : 7.62		
3rd Qu.: 0.00000	3rd Qu.: 0.00		
Max. : 14.00000	Max. : 373.00		
NA's : 35045	NA's : 35045		

Local Machine

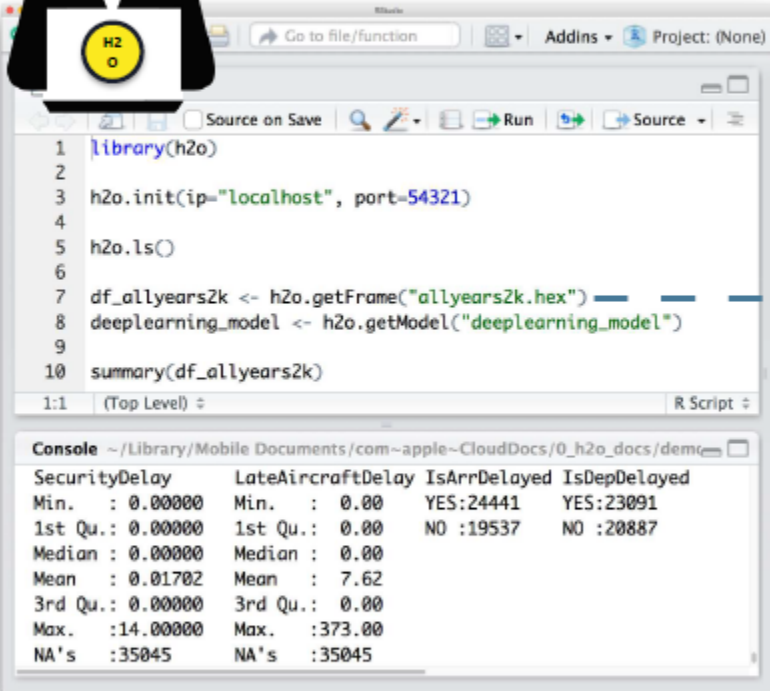



H2O Cluster

REST/
JSON



H2O Clients




```
1 library(h2o)
2
3 h2o.init(ip="localhost", port=54321)
4
5 h2o.ls()
6
7 df_allyears2k <- h2o.getFrame("allyears2k.hex")
8 deeplearning_model <- h2o.getModel("deeplearning_model")
9
10 summary(df_allyears2k)
```

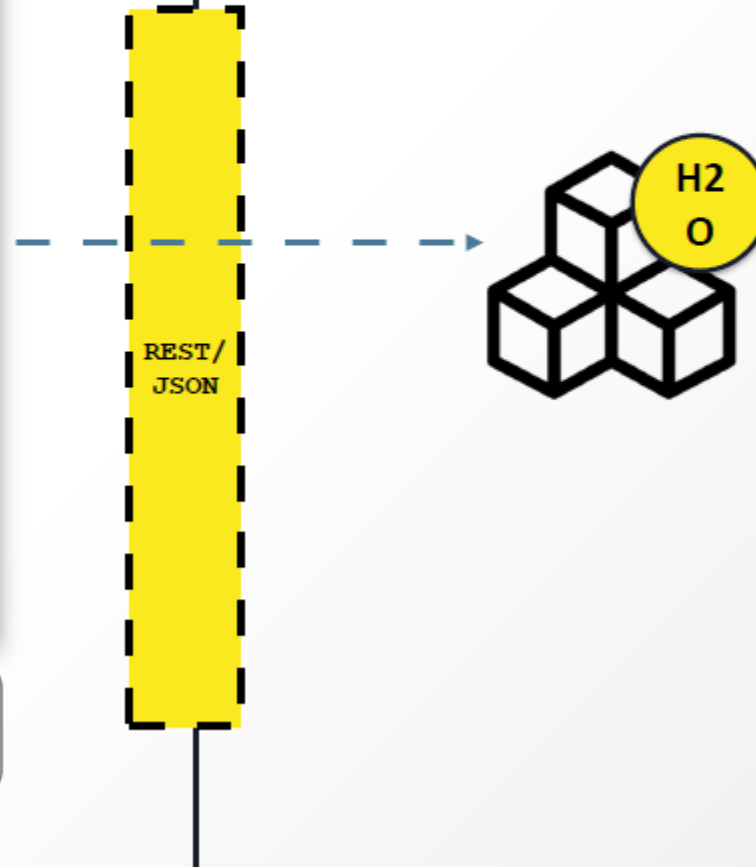
Console ~ /Library/Mobile Documents/com~apple~CloudDocs/0_h2o_docs/dem...

SecurityDelay	LateAircraftDelay	IsArrDelayed	IsDepDelayed
Min. : 0.00000	Min. : 0.00	YES:24441	YES:23091
1st Qu.: 0.00000	1st Qu.: 0.00	NO :19537	NO :20887
Median : 0.00000	Median : 0.00		
Mean : 0.01702	Mean : 7.62		
3rd Qu.: 0.00000	3rd Qu.: 0.00		
Max. : 14.00000	Max. : 373.00		
NA's : 35045	NA's : 35045		

Local Machine



H2O Cluster



H2O Clients

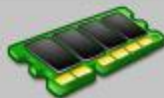


```
1 library(h2o)
2
3 h2o.init(ip="localhost", port=54321)
4
5 h2o.ls()
6
7 df_allyears2k <- h2o.getFrame("allyears2k.hex")
8 deeplearning_model <- h2o.getModel("deeplearning_model")
9
10 summary(df_allyears2k)
```

Console ~/Library/Mobile Documents/com~apple~CloudDocs/0_h2o_docs/demos

SecurityDelay	LateAircraftDelay	IsArrDelayed	IsDepDelayed
Min. : 0.00000	Min. : 0.00	YES:24441	YES:23091
1st Qu.: 0.00000	1st Qu.: 0.00	NO :19537	NO :20887
Median : 0.00000	Median : 0.00		
Mean : 0.01702	Mean : 7.62		
3rd Qu.: 0.00000	3rd Qu.: 0.00		
Max. : 14.00000	Max. : 373.00		
NA's : 35045	NA's : 35045		

Local
Machine



REST/
JSON



H2O Clients

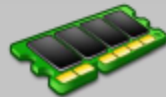


```
library(h2o)
h2o.init(ip="localhost", port=54321)
h2o.ls()
df_allyears2k <- h2o.getFrame("allyears2k.hex")
deeplearning_model <- h2o.getModel("deeplearning_model")
summary(df_allyears2k)
```

Console

SecurityDelay	LateAircraftDelay	IsArrDelayed	IsDepDelayed
Min. : 0.00000	Min. : 0.00	YES:24441	YES:23091
1st Qu.: 0.00000	1st Qu.: 0.00	NO :19537	NO :20887
Median : 0.00000	Median : 0.00		
Mean : 0.01702	Mean : 7.62		
3rd Qu.: 0.00000	3rd Qu.: 0.00		
Max. : 14.00000	Max. : 373.00		
NA's : 35045	NA's : 35045		

Local
Machine



H2O Cluster

REST/
JSON



X ₁	X ₂	X ₃		X _p

H2O Clients



```
1 library(h2o)
2
3 h2o.init(ip="localhost", port=54321)
4
5 h2o.ls()
6
7 df_allyears2k <- h2o.getFrame("allyears2k.hex")
8 deeplearning_model <- h2o.getModel("deeplearning_model")
9
10 summary(df_allyears2k)
```

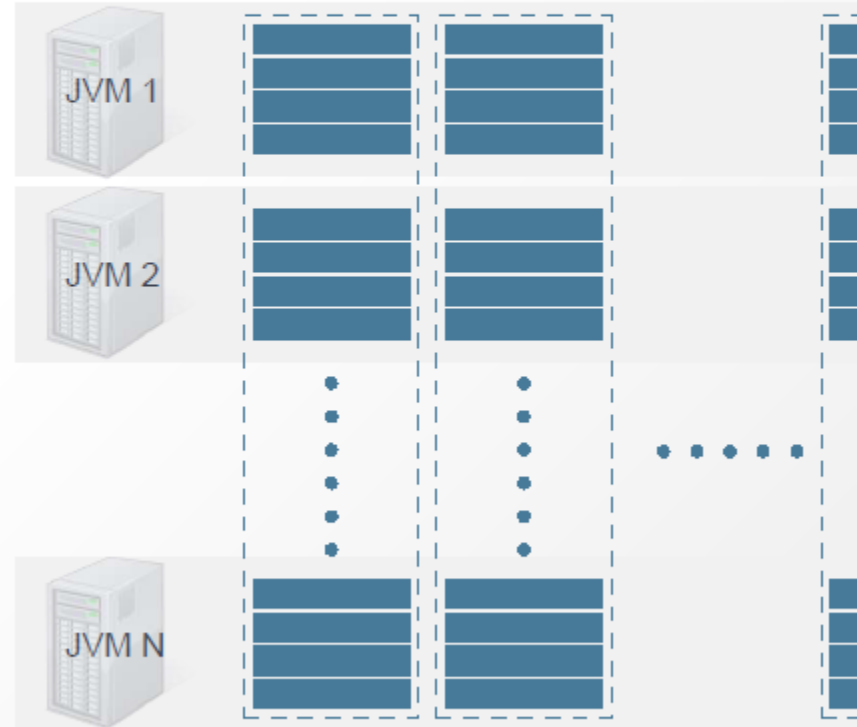
Console ~ /Library/Mobile Documents/com~apple~CloudDocs/0_h2o_docs/dem...

SecurityDelay	LateAircraftDelay	IsArrDelayed	IsDepDelayed
Min. : 0.00000	Min. : 0.00	YES:24441	YES:23091
1st Qu.: 0.00000	1st Qu.: 0.00	NO :19537	NO :20887
Median : 0.00000	Median : 0.00		
Mean : 0.01702	Mean : 7.62		
3rd Qu.: 0.00000	3rd Qu.: 0.00		
Max. : 14.00000	Max. : 373.00		
NA's : 35045	NA's : 35045		

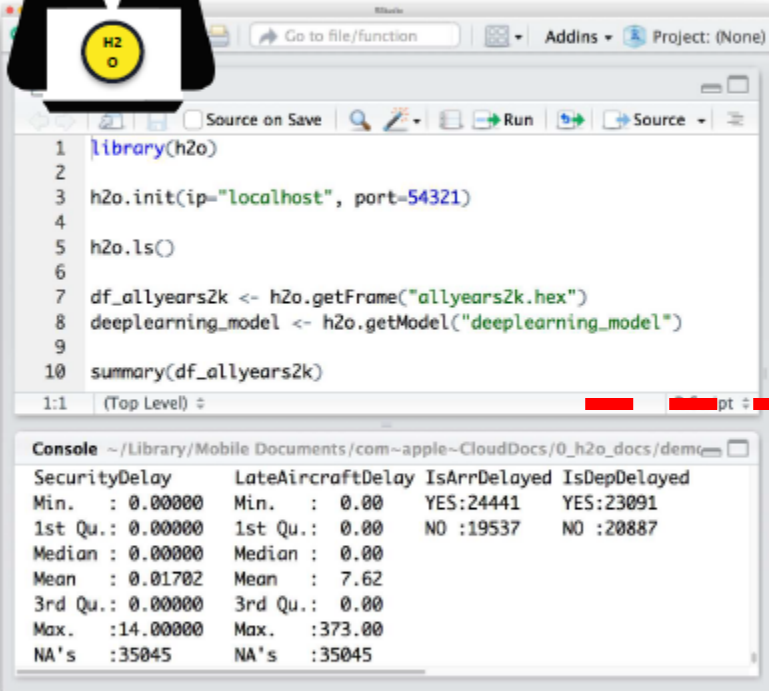

Local Machine

H2O Cluster

REST/
JSON



H2O Clients




```
1 library(h2o)
2
3 h2o.init(ip="localhost", port=54321)
4
5 h2o.ls()
6
7 df_allyears2k <- h2o.getFrame("allyears2k.hex")
8 deeplearning_model <- h2o.getModel("deeplearning_model")
9
10 summary(df_allyears2k)
```

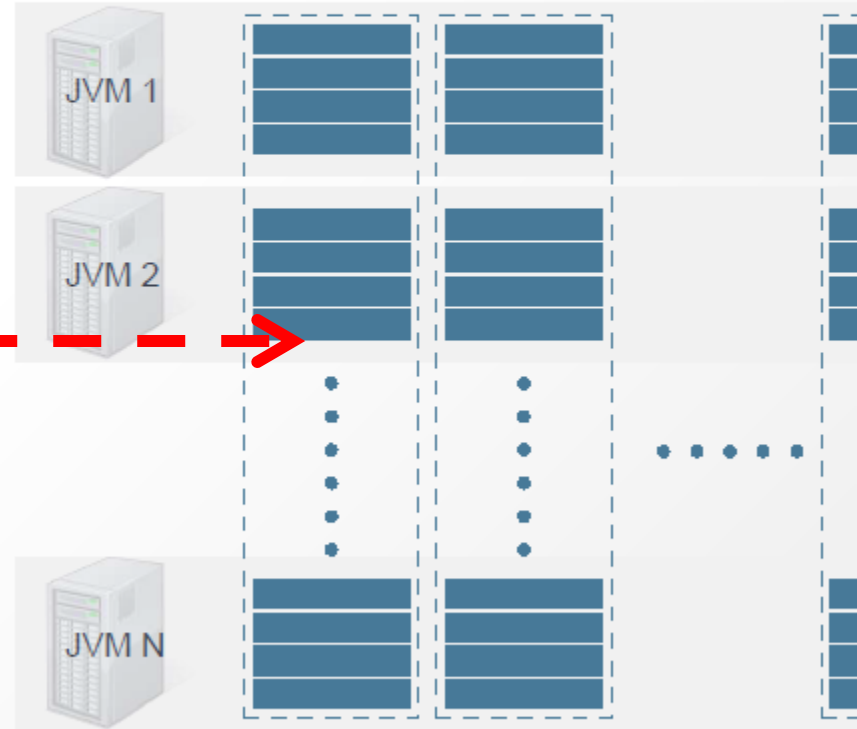
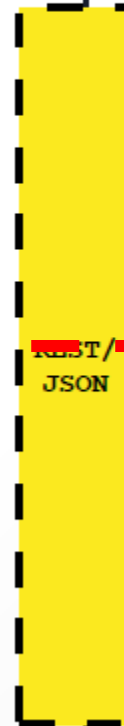
Console ~ /Library/Mobile Documents/com~apple~CloudDocs/0_h2o_docs/demc

SecurityDelay	LateAircraftDelay	IsArrDelayed	IsDepDelayed
Min. : 0.00000	Min. : 0.00	YES:24441	YES:23091
1st Qu.: 0.00000	1st Qu.: 0.00	NO :19537	NO :20887
Median : 0.00000	Median : 0.00		
Mean : 0.01702	Mean : 7.62		
3rd Qu.: 0.00000	3rd Qu.: 0.00		
Max. : 14.00000	Max. : 373.00		
NA's : 35045	NA's : 35045		

Local Machine



H2O Cluster



Mahout Samsara



Samsara es un lenguaje de dominio específico (DSL) para Aprendizaje automático distribuido a gran escala en sistemas como Apache Spark. Características Fundamentales:

- Utiliza Scala como entorno de programación / scripting
- Sistema agnóstico, DSL tipo R. Ejemplo:
$$G = BB^T - C - C^T + \xi^T \xi s_q s_q^T$$

`val G = B %*% B.t - C - C.t + (ksi dot ksi) * (s_q cross s_q)`
- Optimizador de expresiones algebraicas para álgebra lineal distribuida
Proporciona una capa de traducción a los motores distribuidos.

Tipo de Datos DRM (Distributed Row Matrices)

Matriz de Big Data particionada por filas
Reside en la memoria principal del cluster

```
val drmA = drmFromHDFS(...)
```

Operaciones entre matrices, vectores y scalar
in-memory y en forma distribuida

```
drmA %*% drmB  
A %*% x  
A.t %*% drmB  
A * B
```



☐ Resuelve sistemas lineales

```
val x = solve(A, b)
```

☐ descomposiciones de Cholesky, QR, valores singulares y otras:

- ✓ En memoria

```
val (inMemQ, inMemR) = qr(inMemM)
val ch = chol(inMemM)
val (inMemV, d) = eigen(inMemM)
val (inMemU, inMemV, s) = svd(inMemM)
```

✓ Distribuidas

```
val (drmQ, inMemR) = thinQR(drmA)
val (drmU, drmV, s) =
    dssvd(drmA, k = 50, q = 1)
```

☐ almacenamiento en caché de DRM

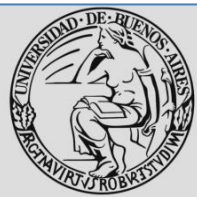
```
val drmA_cached = drmA.checkpoint()

drmA_cached.uncache()
```



Ejemplo

<i>Name</i>	<i>protein</i>	<i>fat</i>	<i>carbo</i>	<i>sugars</i>	<i>rating</i>
Apple Cinnamon Cheerios	2	2	10.5	10	29.509541
Cap'n'Crunch	1	2	12	12	18.042851
Cocoa Puffs	1	1	12	13	22.736446
Froot Loops	2	1	11	13	32.207582
Honey Graham Ohs	1	2	12	11	21.871292
Wheaties Honey Gold	2	1	16	8	36.187559
Cheerios	6	2	17	1	50.764999
Clusters	3	2	13	7	40.400208
Great Grains Pecan	3	3	13	4	45.811716




Carga distribuida del Dataset

1) Lo traigo desde el File system: `val drmData = drmFromHdfs(...)`

2) Lo distribuyo (parallelizo): `val drmData = drmParallelize(dense(`

Name	protein	fat	c
Apple Cinnamon Cheerios	2	2	1
Cap'n'Crunch	1	2	1
Cocoa Puffs	1	1	1
Froot Loops	2	1	1
Honey Graham Ohs	1	2	1
Wheaties Honey Gold	2	1	1
Cheerios	6	2	1
Clusters	3	2	1
Great Grains Pecan	3	3	1



```
(2, 2, 10.5, 10, 29.509541), // Apple Cinnamon Cheerios  
(1, 2, 12, 12, 18.042851), // Cap'n'Crunch  
(1, 1, 12, 13, 22.736446), // Cocoa Puffs  
(2, 1, 11, 13, 32.207582), // Froot Loops  
(1, 2, 12, 11, 21.871292), // Honey Graham Ohs  
(2, 1, 16, 8, 36.187559), // Wheaties Honey Gold  
(6, 2, 17, 1, 50.764999), // Cheerios  
(3, 2, 13, 7, 40.400208), // Clusters  
(3, 3, 13, 4, 45.811716)), // Great Grains Pecan  
numPartitions = 2)
```



Preparación del dataset

Ejemplo de dataset de cereales: la variable objetivo **y** es la calificación del cliente, los pesos de los ingredientes son las variables predictoras **x**

- Segmento (slicing) a X como DRM y le acoplo la respuesta **y** como un vector

```
val drmX = drmData(:, 0 until 4)
```

```
val y = drmData.collect(:, 4)
```

drmX				y
2	2	10.5	10	29.509541
1	2	12	12	18.042851
1	1	12	13	22.736446
2	1	11	13	32.207582
1	2	12	11	21.871292
2	1	16	8	36.187559
6	2	17	1	50.764999
3	2	13	7	40.400208
3	3	13	4	45.811716



Estimación de los coeficientes de la Regresión Múltiple

Método de Mínimos cuadrados ordinarios: minimiza la suma de los cuadrados de los residuos (diferencia entre la verdadera variable objetivo y la predicción del modelo)

- Expresión de forma explícita para la estimación de β como

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

- Computar $X^T X$ y $X^T y$ es tan simple como escribir las fórmulas:

```
val drmXtX = drmX.t %*% drmX
```

```
val drmXty = drmX %*% y
```



Deep Learning en H2O

Funciones de Activación y Pérdida

Siendo: $\alpha = \sum_i w_i x_i + b$, tenemos que las funciones activación y de pérdida más utilizadas en DL son:

Function	Formula	Range
Tanh	$f(\alpha) = \frac{e^{\alpha} - e^{-\alpha}}{e^{\alpha} + e^{-\alpha}}$	$f(\cdot) \in [-1, 1]$
Rectified Linear	$f(\alpha) = \max(0, \alpha)$	$f(\cdot) \in \mathbb{R}_+$

Function	Formula	Typical use
Mean Squared Error	$L(W, B j) = \frac{1}{2} \ t^{(j)} - o^{(j)}\ _2^2$	Regression
Absolute	$L(W, B j) = \ t^{(j)} - o^{(j)}\ _1$	Regression
Huber	$L(W, B j) = \begin{cases} \frac{1}{2} \ t^{(j)} - o^{(j)}\ _2^2 & \text{for } \ t^{(j)} - o^{(j)}\ _1 \leq 1, \\ \ t^{(j)} - o^{(j)}\ _1 - \frac{1}{2} & \text{otherwise.} \end{cases}$	Regression
Cross Entropy	$L(W, B j) = - \sum_{y \in O} \left(\ln(o_y^{(j)}) \cdot t_y^{(j)} + \ln(1 - o_y^{(j)}) \cdot (1 - t_y^{(j)}) \right)$	Classification



Entrenamiento en paralelo

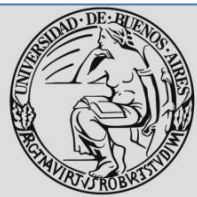
Se utiliza una versión paralelizada del algoritmo de gradiente descendente estocástico. La constante alfa es la tasa de aprendizaje, que controla el tamaño de los pasos incrementales en cada iteración que calcula el gradiente descendente.

El algoritmo tradicional de Gradiente descendente es:

1. Initialize W, B
2. Iterate until convergence criterion reached:
 - a. Get training example i
 - b. Update all weights $w_{jk} \in W$, biases $b_{jk} \in B$

$$w_{jk} := w_{jk} - \alpha \frac{\partial L(W, B | j)}{\partial w_{jk}}$$

$$b_{jk} := b_{jk} - \alpha \frac{\partial L(W, B | j)}{\partial b_{jk}}$$

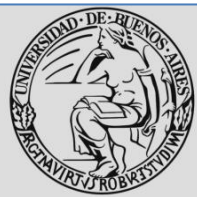


1. Initialize global model parameters W, B
2. Distribute training data \mathcal{T} across nodes (can be disjoint or replicated)
3. Iterate until convergence criterion reached:
 - 3.1. For nodes n with training subset \mathcal{T}_n , do in parallel:
 - a. Obtain copy of the global model parameters W_n, B_n
 - b. Select active subset $\mathcal{T}_{na} \subset \mathcal{T}_n$
 - c. Partition \mathcal{T}_{na} into \mathcal{T}_{nac} by cores n_c
 - d. For cores n_c on node n , do in parallel:
 - i. Get training example $i \in \mathcal{T}_{nac}$
 - ii. Update all weights $w_{jk} \in W_n$, biases $b_{jk} \in B_n$

$$w_{jk} := w_{jk} - \alpha \frac{\partial L(W, B | j)}{\partial w_{jk}}$$

$$b_{jk} := b_{jk} - \alpha \frac{\partial L(W, B | j)}{\partial b_{jk}}$$

- 3.2. Set $W, B := \text{Avg}_n W_n, \text{Avg}_n B_n$



Regularización de Modelos

Los modelos menos flexibles. ser más fáciles de interpretar y tiene mayor capacidad de generalización. Es bien sabido que a más variables el modelo es más flexible y por ende su varianza es mayor. Por ello vamos a analizar métodos de regularización de los modelos de ajuste que quiten flexibilidad al , obteniendo una reducción de la varianza del modelo y la consiguiente mejora en su capacidad de generalización .

Regularización: Regresión Contraída

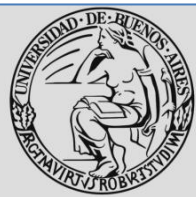
El primer método de regularización o encogimiento (*shrinkage*) es el de Regresión Contraída, o *Ridge Regression* en inglés

En RL queremos minimizar esta funcion por CMín

$$RSS = \sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2$$

En RL Contraída le
adicionamos :

$$\sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^p \beta_j^2 = RSS + \lambda \sum_{j=1}^p \beta_j^2$$



Regresión LASSO

La desventaja de la regresión contraída es que contiene TODOS los pesos. La penalización *ridge* puede llegar a contraer todos los coeficientes β_j a un valor cercano a cero, pero nunca hacerlos llegar exactamente a cero.

En LASSO se reemplaza la penalización $|\beta_j|^2$ por $|\beta_j|$

$$\sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^p |\beta_j| = \text{RSS} + \lambda \sum_{j=1}^p |\beta_j|$$

Esta nueva penalización si consigue llevar algunos coeficientes a **cero**, consiguiendo así un modelo con menor varianza y más fácil de interpretar.

En nomenclatura de DL

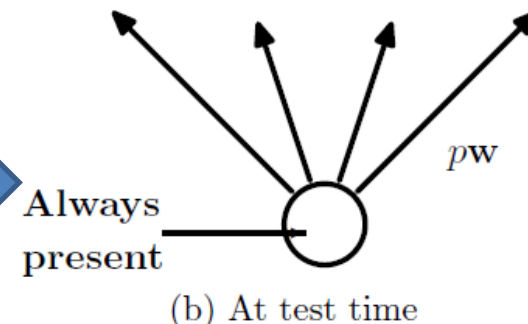
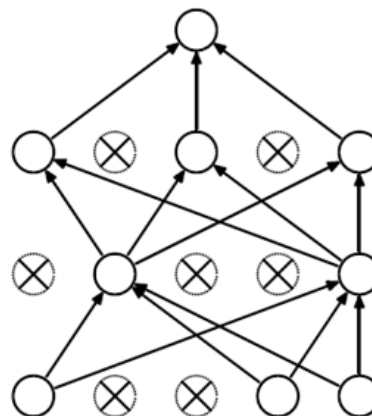
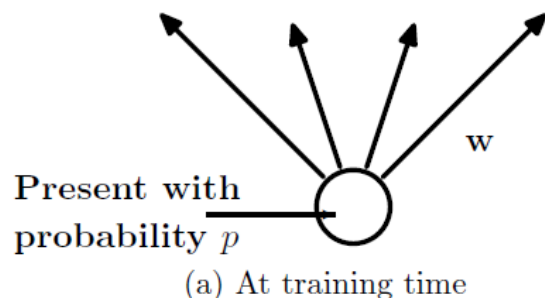
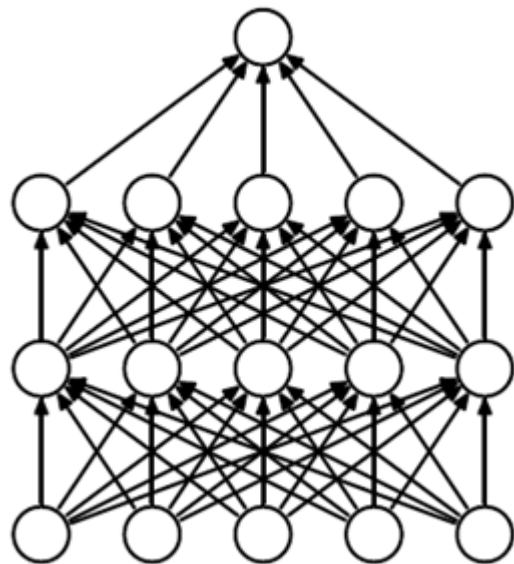
$$L(W, B|j) = L(W, B|j) + \lambda_1 R_1(W, B|j) + \lambda_2 R_2(W, B|j)$$

↑
 ℓ_1

↑
 ℓ_2



Regularización Dropout



Journal of Machine Learning Research 15 (2014) 1929-1958

Submitted 11/13; Published 6/14

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava
Geoffrey Hinton
Alex Krizhevsky
Ilya Sutskever
Ruslan Salakhutdinov

Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

Editor: Yoshua Bengio

NITISH@CS.TORONTO.EDU
HINTON@CS.TORONTO.EDU
KRIZ@CS.TORONTO.EDU
ILYA@CS.TORONTO.EDU
RSALAKHU@CS.TORONTO.EDU



Todos los Parámetros de un Modelo de DL

- **x**: A vector containing the names of the predictors in the model. No default.
- **y**: The name of the response variable in the model. No default.
- **training_frame**: An H2OFrame object containing the variables in the model. No default.
- **model_id**: (Optional) The unique ID assigned to the resulting model. If not specified, an ID is generated automatically.
- **overwrite_with_best_model**: Logical. If enabled, overwrite the final model with the best model that was ever scored during training. Default is true.
- **validation_frame**: (Optional) An H2OFrame object that represents the validation dataset used to construct the confusion matrix. If blank and `n_folds = 0`, the training data is used by default.
- **checkpoint**: (Optional) Model checkpoint (either key or H2ODeepLearningModel) used to resume training.
- **autoencoder**: Logical. Enable autoencoder; default is false. Refer to the "Deep Autoencoders" section for more details.
- **use_all_factor_levels**: Logical. Use all factor levels of categorical variance. Otherwise, the first factor level is omitted (without loss of accuracy). Useful for variable importances and auto-enabled for autoencoder. Default is true.
- **activation**: The choice of nonlinear, differentiable activation function used throughout the network. Options are Tanh, TanhWithDropout, Rectifier, RectifierWithDropout, Maxout, MaxoutWithDropout, and the default is Rectifier. Refer to the "Activation and loss functions" or "???" sections for more details.
- **hidden**: The number and size of each hidden layer in the model. For example, if `c(100,200,100)` is specified, a model with 3 hidden layers will be produced, the middle hidden layer will have 200 neurons and the first and third hidden layers will have 100 neurons. The default is `c(200,200)`. For grid search, use `list(c(10,10), c(20,20))` etc. Refer to the section on "Performing a trial run" for more details.
- **epochs**: The number of passes or over the training dataset (can be fractional). It is recommended to start with lower values for initial grid searches. The value can be modified during checkpoint restarts and allows continuation of selected models. Default is 10.
- **train_samples_per_iteration**: The number of training samples (globally) per MapReduce iteration. The following special values are also supported: 0 (one epoch); -1 (all available data; e.g., replicated training data); -2 (auto-tuning; default). Refer to the "Specifying the number of training samples per iteration" for more details.
- **seed**: The random seed controls sampling and initialization. Reproducible results are only expected with single-threaded operation (i.e. when running on one node, turning off load balancing and providing a small dataset that fits in one chunk). In general, the multi-threaded asynchronous updates to the model parameters will result in (intentional) race conditions and non-reproducible results. Default is a randomly generated number.
- **adaptive_rate**: Logical. Enables adaptive learning rate (ADDELTA). Default is true. Refer to the "Adaptive learning" section for more details.
- **rho**: Adaptive learning rate time decay factor. This parameter is similar to momentum and relates to the memory to prior weight updates. Typical values are between 0.9 and 0.999. Default value is 0.99. Refer to the section on "Adaptive learning" for more details.
- **epsilon**: The second of two hyperparameters for adaptive learning rate (when it is enabled). This parameter is similar to learning rate annealing during initial training and momentum at later stages where it allows forward progress. Typical values are between 1e-10 and 1e-4. This parameter is only

- active if `adaptive_rate` is enabled. Default is 1e-8. Refer to the "Adaptive learning" section for more details.
- **rate**: The learning rate, α . Higher values lead to less stable models, while lower values lead to slower convergence. Default is 0.005.
- **rate_annealing**: The annealing learning rate is calculated as $(rate)/(1+rate_annealing * N)$, where N is the number of training samples. It is the inverse of the number of training samples required to cut the learning rate in half. This reduces the learning rate to "frsize" into local minima in the optimization landscape. Default value is 1e-6 (when adaptive learning is disabled). Refer to the "Rate annealing" section for more details.
- **rate_decay**: Learning rate decay factor between layers (L-th layer: $rate * rate_decay^{(L-1)}$); default is 1.0 (when adaptive learning is disabled). The learning rate decay parameter controls the change of learning rate across layers.
- **momentum_start**: This controls the amount of momentum at the beginning of training (when adaptive learning is disabled). Default is 0. Refer to the "Momentum training" section for more details.
- **momentum_ramp**: This controls the amount of learning for which momentum increases (assuming `momentum_stable` is larger than `momentum_start`). The ramp is measured in the number of training samples and can be enabled when adaptive learning is disabled. Default is 1 million (1e6). Refer to the "Momentum training" section or more details.
- **momentum_stable**: This controls the final momentum value reached after `momentum_ramp` number of training samples (when adaptive learning is disabled). The momentum used for training will remain the same for training beyond reaching that point. Default is 0. Refer to the "Momentum training" for more details.
- **nesterov_accelerated_gradient**: Logical. The Nesterov accelerated gradient descent method is a modification to traditional gradient descent for convex functions. The method relies on gradient information at various points to build a polynomial approximation that minimizes the residuals in fewer iterations of the descent. This parameter is only active if adaptive learning rate is disabled. The default is true (when adaptive learning is disabled). Refer to the "Momentum training" section for more details.
- **input_dropout_ratio**: The fraction of the features for each training row to be omitted from training in order to improve generalization. The default is 0 (always use all features). Refer to the "Regularization" section for more details.
- **hidden_dropout_ratio**: The fraction of the inputs for each hidden layer to be omitted from training in order to improve generalization. The default is 0.5 for each hidden layer. Refer to the "Regularization" section for more details.
- **l1**: L1 regularization, which constrains the absolute value of the weights (can add stability and improve generalization, causes many weights to become 0). The default is 0, for no L1 regularization. Refer to the "Regularization" section for more details.
- **l2**: L2 regularization, which constrains the sum of the squared weights (can add stability and improve generalization, causes many weights to be small). The default is 0, for no L2 regularization. Refer to the "Regularization" section for more details.
- **max_w2**: A maximum on the sum of the squared incoming weights into any one neuron. This tuning parameter is especially useful for unbound activation functions such as Maxout or Rectifier. The default, positive infinity, leaves this maximum unbounded.
- **initial_weight_distribution**: The distribution from which initial weights are to be drawn. Select Uniform, UniformAdaptive or Normal. Default is UniformAdaptive. Refer to the "Initialization" for more details.
- **initial_weight_scale**: The scale of the distribution function for Uniform or Normal distributions. For Uniform, the values are drawn uniformly from $(-initial_weight_scale, initial_weight_scale)$.



For Normal, the values are drawn from a Normal distribution with a standard deviation of `initial_weight`. The default is 1. Refer to the "Initialization" for more details.

- **loss**: Specify one of the loss options: Automatic, CrossEntropy (for classification only), MeanSquare, Absolute, or Huber. Refer to the "Activation and loss functions" section for more details.
- **distribution**: Specify the distribution function of the response: AUTO, bernoulli, multinomial, poisson, gamma, tweedie, laplace, huber, or gaussian.
- **tweedie_power**: Specify the tweedie power; applicable only if **distribution** is set to tweedie. Value must be between 1.0 and 2.0.
- **score_interval**: The minimum time (in seconds) to elapse between model scoring. The actual interval is determined by the number of training samples per iteration and the scoring duty cycle. To use all training set samples, specify 0. Default is 5.
- **score_training_samples**: The number of training samples to be used for scoring. These samples will be randomly sampled. Use 0 to select the entire training dataset. Default is 10000.
- **score_validation_samples**: The number of validation dataset points to be used for scoring. Can be randomly sampled or stratified (if **balance_classes** is set and **score_validation_sampling** is set to stratify). Use 0 to select the entire validation dataset (default).
- **score_duty_cycle**: Maximum duty cycle fraction spent on model scoring (on both training and validation samples), and on diagnostics such as computation of feature importances (i.e., not on training). Lower values result in more training, while higher values produce more scoring. Default is 0.1.
- **classification_stop**: The stopping criterion for classification error (1 - accuracy) on the training data scoring dataset. When the error is at or below this threshold, the training process stops. Default is 0. To disable, enter -1.
- **regression_stop**: The stopping criterion for regression error (MSE) on the training data scoring dataset. When the error is at or below this threshold, the training process stops. Default is 1e-6. To disable, enter -1.
- **quiet_mode**: Logical. Enable quiet mode for less output to standard output. Default is false.
- **max_confusion_matrix_size**: For classification models, the maximum size (in terms of classes) of the confusion matrix to display. This option is meant to avoid printing extremely large confusion matrices. Default is 20.
- **max_hit_ratio_k**: The maximum number (top K) of predictions to use for hit ratio computation (for multi-class only, enter 0 to disable). Default is 10.
- **balance_classes**: Logical. For imbalanced data, the training data class counts can be artificially balanced by over-sampling the minority class(es) and under-sampling the majority class(es) so that each class will contain the same number of observations. This can result in improved predictive accuracy. Over-sampling is done with replacement (rather than simulating new observations), and the total number of observations after balancing is controlled by the **max_after_balance_size** parameter. Default is false.
- **class_sampling_factors**: Desired over/under-sampling ratios per class (in lexicographic order). Only applies to classification when **balance_classes** is enabled. If not specified, the ratios will be automatically computed to obtain class balance during training.
- **max_after_balance_size**: When classes are balanced, limit the resulting dataset size to the specified multiple of the original dataset size. This is the maximum relative size of the training data after balancing class counts (can be less than 1.0). Default is 5.
- **score_validation_sampling**: Method used to sample validation dataset for scoring. The possible methods are Uniform and Stratified. Default is Uniform.

- **diagnostics**: (Deprecated) Logical. Gather diagnostics for hidden layers, such as mean and root mean squared (RMS) values of learning rate, momentum, weights and biases. Since deprecation, diagnostics are always enabled (set to true).
- **variable_importance**: Logical. Compute variable importances for input features using the Gadeon method. The implementation considers the weights connecting the input features to the first two hidden layers. Default is false, since this can be slow for large networks.
- **fast_mode**: Logical. Enable fast mode (minor approximation in back-propagation). This should not affect results significantly. Default is true.
- **ignore_constant_cols**: Logical. Ignore constant training columns (no information can be gained anyway). Default is true.
- **force_load_balance**: Logical. Force extra load balancing to increase training speed for small datasets to keep all cores busy. Default is true.
- **replicate_training_data**: Logical. Replicate the entire training dataset onto every node for faster training on small datasets. Default is true.
- **single_node_mode**: Logical. Run on a single node for fine-tuning of model parameters. Can be useful for faster convergence during checkpoint resumes after training on a very large count of nodes (for fast initial convergence). Default is false.
- **shuffle_training_data**: Logical. Enable shuffling of training data (on each node). This option is recommended if training data is replicated on N nodes, and the number of training samples per iteration is close to N times the dataset size, where all nodes train with (almost) all of the data. It is automatically enabled if the number of training samples per iteration is set to -1 (or to N times the dataset size or larger). Default is false.
- **sparse**: (Deprecated) Logical. Enable sparse data handling. Default is false.
- **col_major**: (Deprecated) Logical. Use a column major weight matrix for the input layer; can speed up forward propagation, but may slow down backpropagation. Default is false.
- **average_activation**: Specify the average activation for the sparse autoencoder (Experimental). Default is 0.
- **sparsity_beta**: Specify the sparsity-based regularization optimization (Experimental). Default is 0.
- **max_categorical_features**: Maximum number of categorical features allowed in a column, enforced via hashing (Experimental). Default is $2^{31} - 1$ (Integer.MAX_VALUE in Java).
- **reproducible**: Logical. Force reproducibility on small data (will be slow - only uses one thread). Default is false.
- **export_weights_and_biases**: Logical. Specify whether to export the neural network weights and biases as an H2OFrame. Default is false.
- **offset_column**: Specify the offset column by column name. Regression only. Offsets are per-row "bias values" that are used during model training. For Gaussian distributions, they can be seen as simple corrections to the response (y) column. Instead of learning to predict the response value directly, the model learns to predict the (row) offset of the response column. For other distributions, the offset corrections are applied in the linearized space before applying the inverse link function to get the actual response values.
- **weights_column**: Specify the weights column by column name. Weights are per-row observation weights. This is typically the number of times a row is repeated, but non-integer values are supported as well. During training, rows with higher weights matter more, due to the larger loss function pre-factor.
- **nfold**: (Optional) Number of folds for cross-validation. Default is 0 (no cross-validation is performed).



Hiperparámetros para Modelos de RNA con H2O

Interrupción del Entrenamiento

- **Early Stopping:** H2O evalúa regularmente la performance predictiva del modelo en el set validación, validación cruzada y / o set de entrenamiento y a partir de ellos evalúa la detención del entrenamiento del modelo
 - *stopping_metric*: “logloss” para clasificación y “deviance” para regresión
 - *stopping_tolerance*: Detener si (la métrica elegida en *stopping_metric*:) no ha mejorado al menos hasta lo indicado por este parámetro.
 - *stopping_rounds*: Los gráficos de historial de performance a veces se tambalean un poco. Estableciendo este parámetro en un valor mayor a 1 se deja espacio para que las métricas “deambulen” antes de que lleguen a su mejor valor. Funciona al comparar dos promedios móviles; lo más temprano que puede detenerse es después de dos veces este número de rondas de puntuación. Elegir 1 significa que el modelo tiene que mejorar en cada ronda de puntuación individual. Establézcalo en cero para no usar **early_stopping** nunca..



Medición de la la performance (scoring) del Modelo

- *train_samples_per_iteration*: Esto controla cuántas filas de entrenamiento (muestras) usar por iteración; una iteración se puede pensar como el tiempo entre diferentes mediciones de la performance del modelo. Es un número entero, y normalmente se utiliza uno de los siguientes tres valores especiales. -2 es el valor predeterminado, y deja H2O decidir. 0 y -1 significan lo mismo: una época
- *replicate_training_data*: El valor predeterminado es verdadero. Si es verdadero, replicará todo el conjunto de datos de entrenamiento en cada nodo del cluster. Para pequeños conjuntos de datos, esto puede dar como resultado un entrenamiento más rápido
- *score_validation_samples*: Cuantas de las filas del set de validación voy a usar para medir la performance del modelo. El valor predeterminado de 0 significa usarlos a todos. Si el set validación es grandes o se están obteniendo puntajes con mucha frecuencia, es posible que desee elegir un número más bajo para acelerar la puntuación (con riesgo de una posible reducción de la exactitud (accuracy) de la predicción)
- *score_training_samples*: Al igual que *score_validation_samples*, pero aplicado a las filas de de datos del set de entrenamiento. El valor predeterminado es 10,000, para asegurarse de que no sea muy grande la cantidad de datos que se utiliza en cada medición de scoring, con su consiguiente riesgo de ralentizar el entrenamiento del modelo.



Ejemplo de scoring: clúster de 3 nodos, 120,000 filas de entrenamiento, 40,000 por nodo.

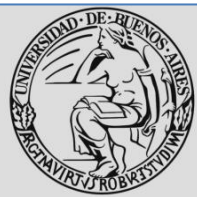
Hiperparámetro de interrupción temprana *stopping_rounds* default=5

Proceso:

- Cada nodo empieza a entrenar su propio modelo con sus 40,000 observaciones, actualizándose en cada pasada por la red (forward y back-propagation) los valores de los pesos y los sesgos de cada neurona activa (weight w y bias b). En este proceso no hay comunicación alguna con los demás nodos del cluster (o sea con los dos modelos restantes que se entrenan con los otros 80.000 valores del data set de entrenamiento)
- Sigue el proceso recién descrito durante una iteración, cuya duración es controlada por el hiper-parámetro *train_samples_per_iteration*. Si utilizo el valor predeterminado de -2 lo decidirá H2O, pero generalmente será una fracción de una época para grandes conjuntos de datos, y podría incluir varias épocas para pequeños conjuntos de datos en cada iteración.
- Al final de la iteración, cada nodo se detendrá y compartirá sus ponderaciones / sesgos con cada los demás nodos. Se promedian (de modo que cada nodo ahora tiene pesos idénticos!). A su vez se medirá la performance del modelo hasta allí (*checkpoint*) de la siguiente manera:
 - **Train:** Se calcularán las métricas de performance en 3333 datos de los 40000 que se utilizaron para entrenar el modelo que arrojó cada nodo (esto es porque el hiper-parámetro *score_training_samples* tiene su valor por default en 10000)
 - **Test/Val:** Se calcularán las métricas de performance en la totalidad del set de validación, pues el hiper-parámetro *score_validation_samples* tiene su valor por default en 0)



- Se carga un nuevo dato al *scoring* del modelo general . Si es el mejor modelo entonces hasta ahora, se guarda una instantánea del modelo. Y luego cada nodo comienza a entrenar nuevamente sus propias 40.000 filas
- Al final de la segunda iteración, los pesos / sesgos son promediados y compartidos nuevamente. El algoritmo considerará la posibilidad de detenerse (*early stopping*) por primera vez después de 10 iteraciones pues precisará evaluar dos cantidades: el promedio de la métrica de puntuación para las rondas 1 a 5, (media móvil 5, MM1-5) y el promedio de la métrica de puntuación para las rondas 6 a 10 (MM6-10 también), Si la diferencia entre MM6-10 y MM1-5 es menor a *stopping_tolerance* entonces de interrumpirá (*early stopping*). De lo contrario continuará comparando el promedio de MM2-6 vs MM7-11.Y así sucesivamente.
- Sobre el hiper-parámetro *train_samples_per_iteration* Si su valor es **0**, cada nodo haría 40,000 muestras por iteración. Si es **-1**, cada nodo haría exactamente 120,000 muestras por iteración., pues es el total de training set.



Para mejorar la generalización

- **Drop-out:** elimina neuronas y todas sus conexiones
- **Regularización:** reduce ciertos pesos de neuronas, o directamente los elimina (L2 o L1)

Veamos diferentes parámetros que debemos definir para ir mejorando la capacidad de generalización del modelo de redes neuronales a implementar:

- *activation* : Este parámetro tiene seis valores posibles. Primero se puede elegir entre tres funciones de activación : Rectifier, Tanh y Maxout. Luego se puede optar por drop-out o no. Si se quiere usar *hidden_dropout_ratios* se debe especificar:
"TanhWithDropout", "RectifierWithDropout" o "MaxoutWithDropout".
- *hidden_dropout_ratios*: Se especifica una probabilidad por cada capa oculta. El valor por defecto de 0.5 significa para cada fila de entrenamiento que se procesa a través de la red, hay un 50% de probabilidad de que una neurona se active , y un 50% de probabilidad de que no.
- */l1*: Regularización L1. También conocido como regularización LASSO. El valor predeterminado es 0, y valores típicos a probar son 0.0001 o menores.
- */l2*: Regularización L2. También conocido como regularización *ridge* . Valores idem L1

