

Masterclass

by IDATHA Academy

AI

KEEP
LEARNING

DS

Chapter

DL

Chapter

CV

Chapter

Contenido

AGENDA

REPASO HISTÓRICO

I REPASO MACHINE LEARNING

II CONCEPTOS BÁSICOS DEEP LEARNING

III TEOREMA DE APROXIMACIÓN UNIVERSAL

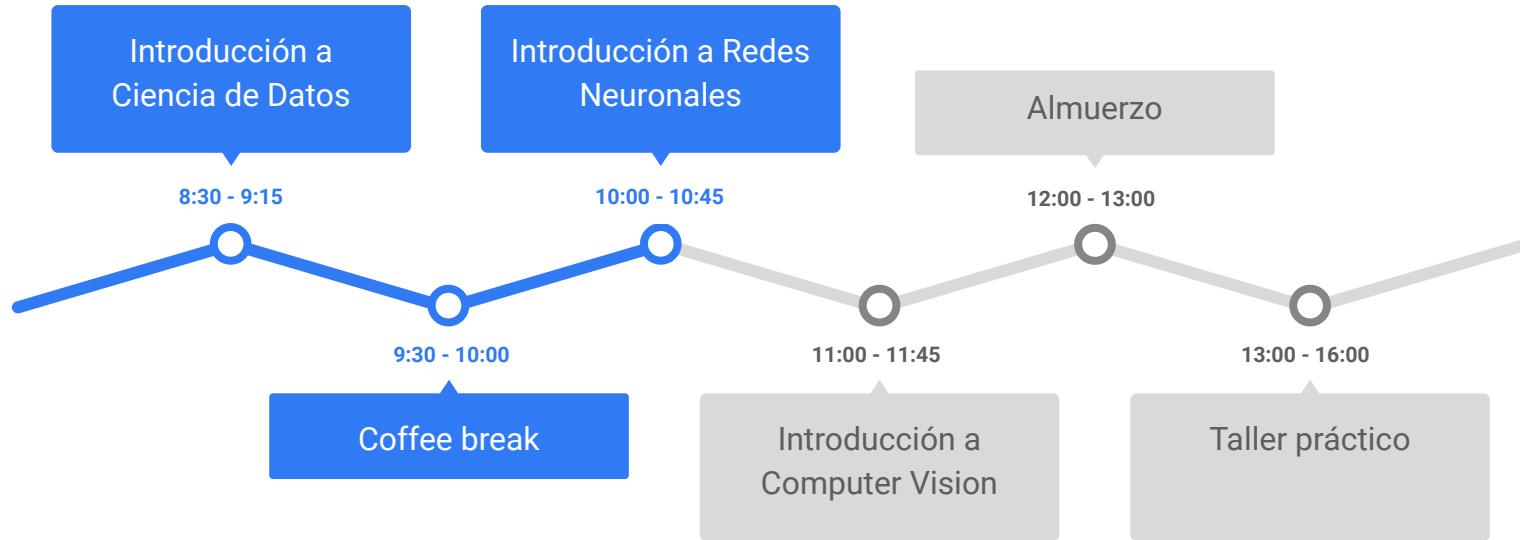
IV GRADIENT DESCENT Y BACK/FORWARD PROPAGATION

V REGULARIZACIÓN Y OPTIMIZACIÓN

VI HERRAMIENTAS Y BIBLIOTECAS



AGENDA



Algunas menciones

Menciones y agradecimientos a los siguientes cursos que fueron tomados como referencia para la elaboración de esta clase:

1. *CS231n: Deep Learning for Computer Vision*
<http://cs231n.stanford.edu/>



2. *Aprendizaje Profundo para Visión Artificial (DLvis), IIE - Fing*
<https://eva.fing.edu.uy/course/view.php?id=1046>



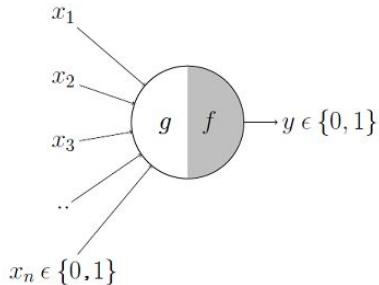
3. *Deep Learning Specialization Andrew NG DeepLearning.ai*
<https://wwwdeeplearning.ai/program/deep-learning-specialization/>



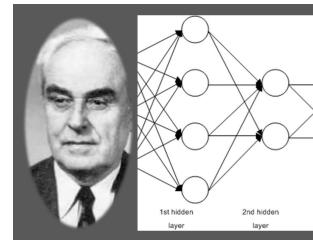
REPASO HISTÓRICO



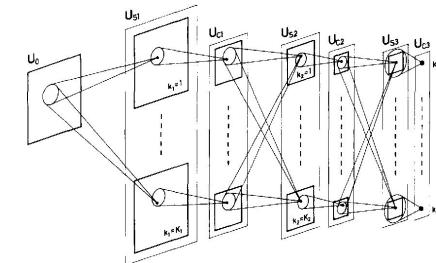
HISTORIA



"Neuronas" de McCulloch & Pitts.



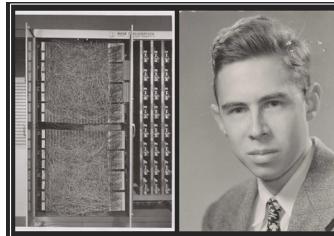
Primera Red Multicapa de Ivakhnenko



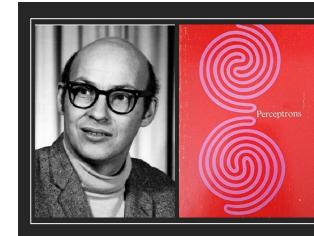
"Neocognitron" de Fukushima



"Perceptron" de Rosenblatt



"Perceptrons" de Minsky y Papert



"Backpropagation" Hinton, Rumelhart, Williams



Learning representations by back-propagating errors

David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams*

* Institute for Cognitive Science, C-410, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, USA

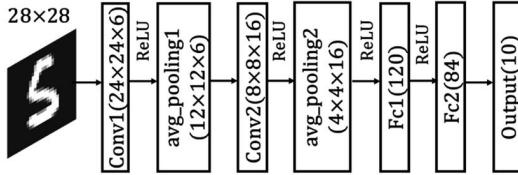
We describe a new learning procedure, back-propagation, for networks of parallel units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output of the network and the desired state vector of the output units. Each iteration involves a propagation of the error term from the output back through the network to the input. During this propagation, the adjustments are proportional to the local weighted partial derivative of the error function with respect to the weights of the connections. The adjustments are small if the error function is smooth, and the regularities in the task are exploited by the interactions between the units. We show that the back-propagation algorithm quickly learns difficult problems that other learning methods do not learn at all. The algorithm also provides a way of decomposing back-propagation from earlier, simpler methods such as gradient descent.

There have been many attempts to design self-organizing neural networks. Most of these attempts have used a gradient modification rule that will allow an arbitrarily connected neural network to learn a set of input-output associations. However, the gradient modification rule is slow, especially for going from a random initial state to a desired state vector of the output units. Such a slow rate of convergence is undesirable because it makes it relatively easy to find learning rules that are good for some tasks but bad for others. In contrast, back-propagation reduces the difference between the actual and the target output much faster. Learning becomes more interesting for

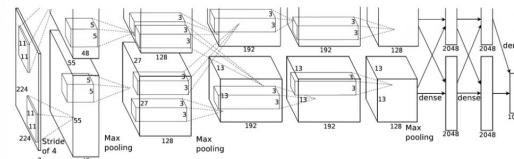
* To whom correspondence should be addressed.



HISTORIA



"CNN" con backpropagation Y.
Lecun.



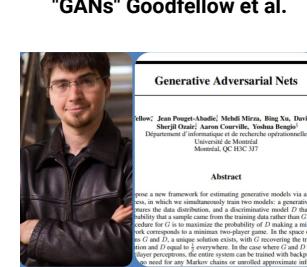
"AlexNet"



"AlphaGo" Silver et al

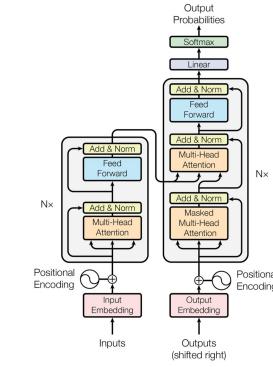


GPUs, ImageNet



"GANs" Goodfellow et al.

"Transformer" Vaswani et al.



PARTE I

1. Repaso machine learning



“Aprendizaje automático se ocupa del construir algoritmos de computación que permiten a programas mejorar su desempeño automáticamente a través de experiencia”.

Tom Mitchell, Machine LEarning, McGraw Hill, 1997



1. Aprendizaje Supervisado

- Predecir una salida dada una entrada.
- Regresión (continuo) o Clasificación (discreto)
- Se necesitan datos de entrenamiento con la salida especificada

2. Aprendizaje NO Supervisado

- Encontrar una buena representación de los datos.
- Encontrar estructura, patrones (explorar los datos)

3. Aprendizaje por refuerzo o recompensa

- Aprender a elegir acciones con el fin de maximizar alguna noción de recompensa acumulada.
- El programa interactúa con un entorno dinámico



Observamos un conjunto de pares $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ donde $x \in \mathbb{R}^N$.

Regresión: La salida deseada es un número real $y \in \mathbb{R}$ o un vector de números reales $y \in \mathbb{R}^m$.

- Predecir la temperatura al mediodía del jueves
- Predecir el valor de la nafta en 6 meses

Clasificación: La salida deseada es una variable categórica, $y \in \{1, \dots, k\}$.

- El caso más sencillo es clasificación binaria, $k = 2$.
- Ejemplos: clasificación de imágenes, reconocimiento de voz.



Observamos un conjunto de pares $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ donde $x \in \mathbb{R}^N$

Proceso de aprendizaje:

- Una familia de funciones $f \in \mathcal{F}$ (predicción)
- Una medida discrepancia (o ajuste), $L(f(x), y)$, entre predicciones $f(x)$ y objetivos y .
- Método para elegir $f^* \in \mathcal{F}$ de forma de:

$$f^* = \arg \min_{f \in \mathcal{F}} \sum_{i=1}^n L(f(\mathbf{x}_i), y_i)$$

f es la función que estamos aprendiendo

Suma de los errores en cada ejemplo del dataset.

Error en la aproximación

En esta clase:

- \mathcal{F} es una familia paramétrica de funciones, $f(x; \theta) = f_\theta(x)$
- Aprendizaje se reduce a resolver el problema de optimización,

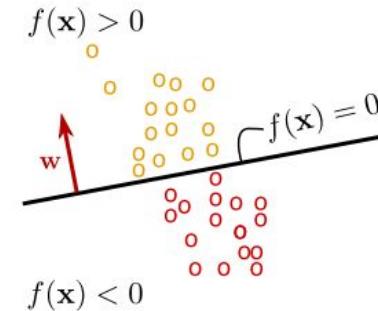
$$\theta^* = \arg \min_{\theta} \sum_{i=1}^n L(f_\theta(\mathbf{x}_i), y_i)$$



CLASIFICADORES LINEALES

Un clasificador lineal tiene la forma:

$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$$



- a $\mathbf{w} \in \mathbb{R}^d$ se lo conoce como vector de pesos (“weights”)
 \mathbf{w} : vector normal a la frontera de clasificación
- a $b \in \mathbb{R}^d$ se le llama “bias” (de-centrado, sesgo)
- Generalización a clasificación multiclase: (\mathbf{w}_i, b_i) , para $i = 1, \dots, c$
(conjunto de pesos por clase)

CLASIFICADORES LINEALES

Definida la forma que tiene la función a ajustar, falta definir la forma en que vamos a medir el error:

SVM (*hinge*)

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Logistic regression (*softmax*)

$$L_i = -\log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

- En ambos casos hay que minimizar $L(W) + \text{regularización}$ (opcional)

$$L(W) = \sum_{i=1}^n L_i(W) + \lambda R(W)$$

- Descenso por gradiente:

$$\nabla_W L(W) = \sum_{i=1}^n \nabla_W L_i(\mathbf{x}_i, y_i; W) + \lambda \nabla_W R(W)$$



CLASIFICADORES LINEALES

Definida la forma que tiene la función a ajustar, falta definir la forma en que vamos a medir el error:

SVM (*hinge*)

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Logistic regression (*softmax*)

$$L_i = -\log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

- En ambos casos hay que minimizar $L(W) + \text{regularización}$ (opcional)

$$L(W) = \sum_{i=1}^n L_i(W) + \lambda R(W)$$

- Descenso por gradiente estocástico, $n_{mb} << n$:

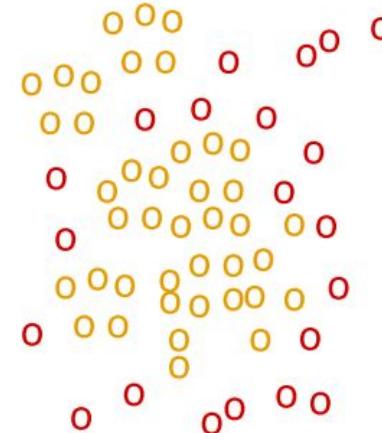
$$\nabla_W L(W) \approx \sum_{i=1}^{n_{mb}} \nabla_W L_i(\mathbf{x}_i, y_i; W) + \lambda \nabla_W R(W)$$



CLASIFICADORES NO LINEALES

Se requiere que los datos sean linealmente separables

$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$$



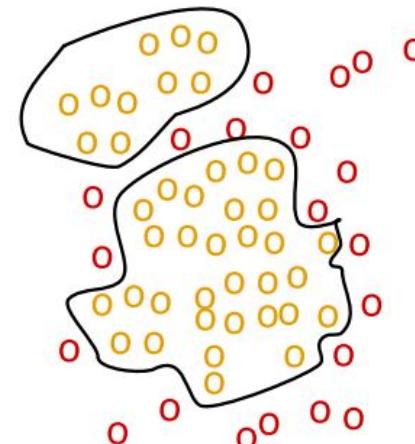
CLASIFICADORES NO LINEALES

Se requiere que los datos sean linealmente separables

→ ¿Cómo encontramos $\Phi(x)$ que transforme los datos a un espacio donde sean fácilmente separables?

$$f(\mathbf{x}) = \mathbf{w}^T \Phi(\mathbf{x}) + b$$

1. Usar $\Phi(x)$ muy genérica que lleve los datos a un espacio de **muy alta dimensión** (puede ser infinita), en este espacio es fácil separar los puntos del conjunto de entrenamiento (e.g *kernel RBF*)
Problema: Difícil de generalizar bien a datos no conocidos (test).



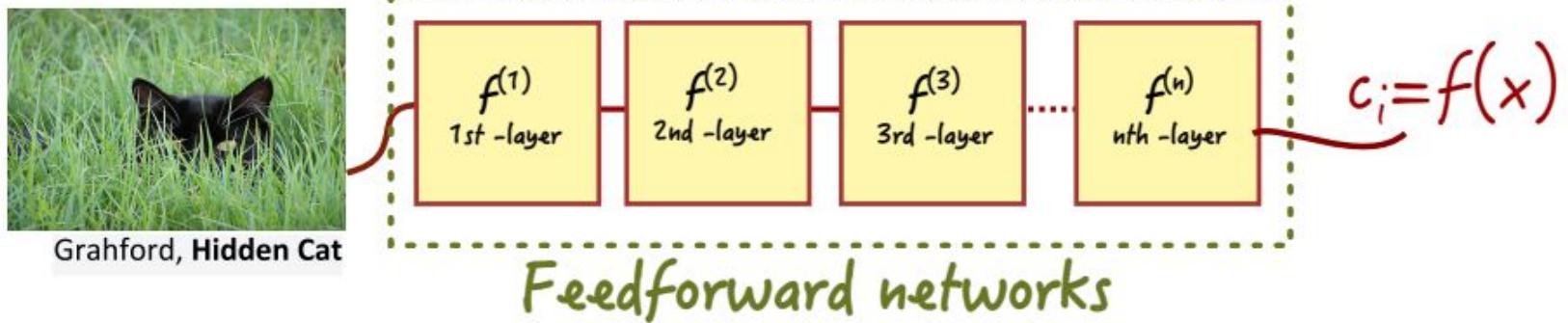
CLASIFICADORES NO LINEALES

Se requiere que los datos sean linealmente separables

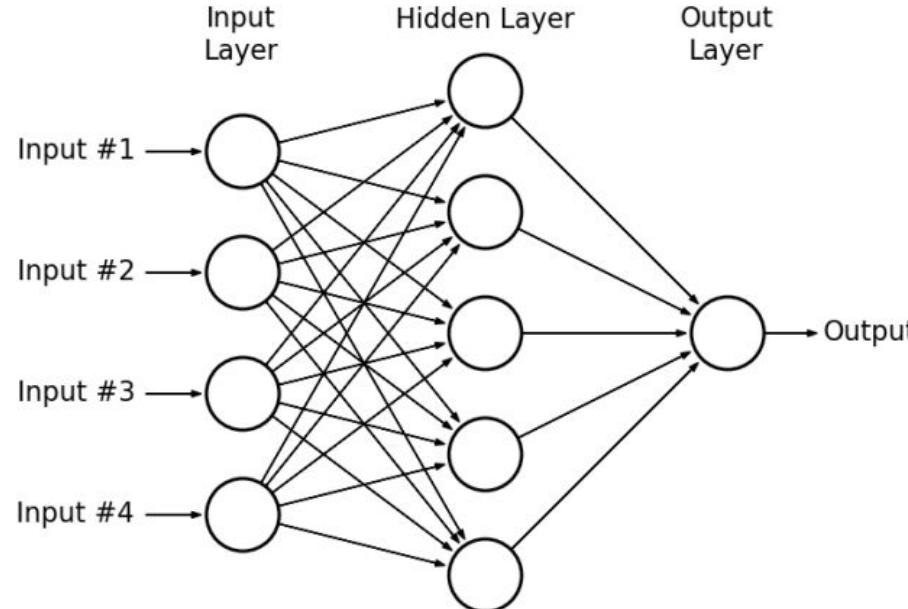
→ ¿Cómo encontramos $\Phi(x)$ que transforme los datos a un espacio donde sean fácilmente separables?

2. Aprendizaje Profundo: Aprender $\Phi(x)$. Partir de un modelo paramétrico $\Phi(x; \theta)$ que define una representación y buscar minimizar el error de ajuste.

Objetivo: Definir $\Phi(x; \theta)$ y luego entrenar (encontrar θ).



- **Objetivo:** Aproximar $y = f^*(x)$, mapeo de x en una salida y (e.g., una categoría).
- **Redes neuronales:** familia de funciones **no lineales**, paramétricas, son la base fundamental del Aprendizaje Profundo.
- Una red neuronal define un mapeo $\hat{y} = f(x; \theta)$, y buscaremos los parámetros θ que mejor aproximan f^* .
- Son redes porque se componen de múltiples neuronas artificiales que se comunican entre sí



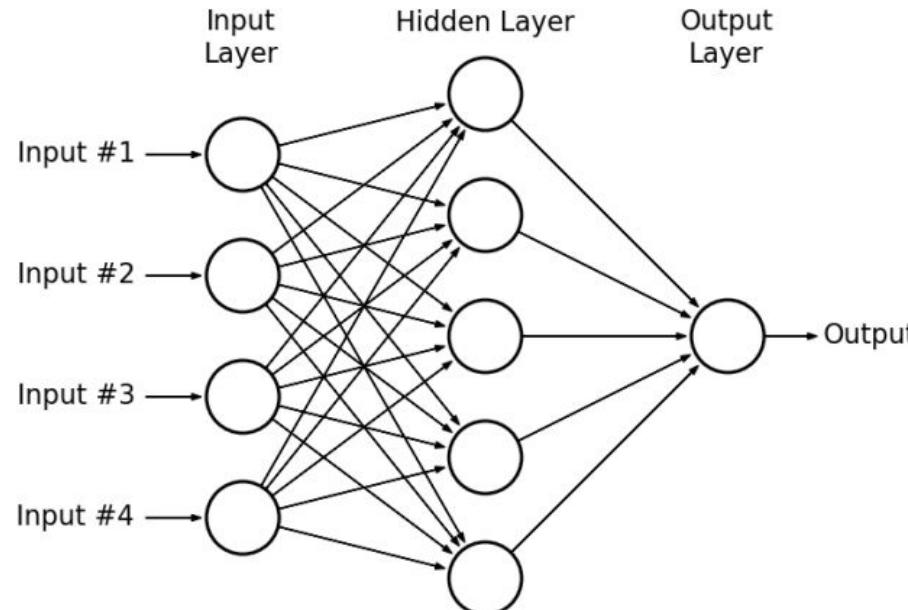
REDES NEURONALES

- En general se componen de varias capas, y se modelan como una composición de funciones:

$$f(x) := f^{(n)} \left(f^{(n-1)} \left(\dots f^{(2)} \left(f^{(1)}(x) \right) \dots \right) \right),$$

donde $f^{(1)}$ es la primera capa (first layer), $f^{(2)}$ la segunda ...

- Son **prealimentadas** (feedforward) porque la información fluye a partir de x sin existir conexiones de realimentación.
- Es decir, son **Grafos Acíclicos Direccionalizados** (DAGs)



PARTE II

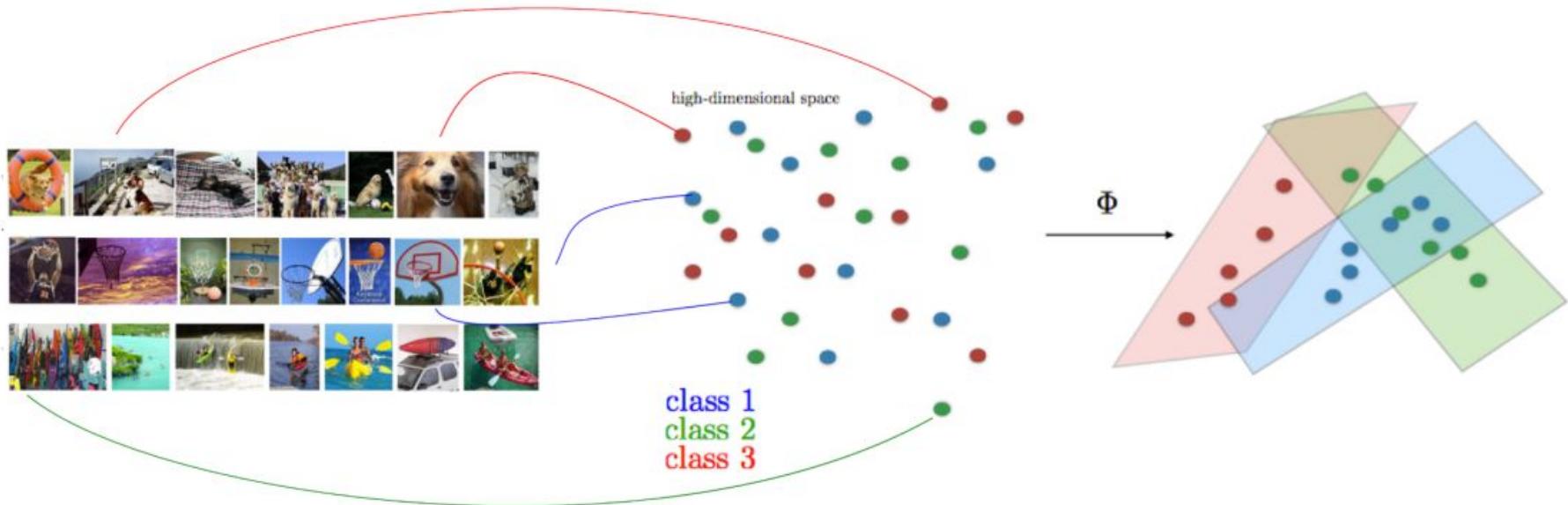
2. Deep learning definiciones



¿Qué es el Aprendizaje Profundo? (Deep Learning)

Clase de representaciones paramétricas no lineales capaces de codificar las características del problema y de ser optimizadas de forma eficiente usando métodos de optimización estocástica.

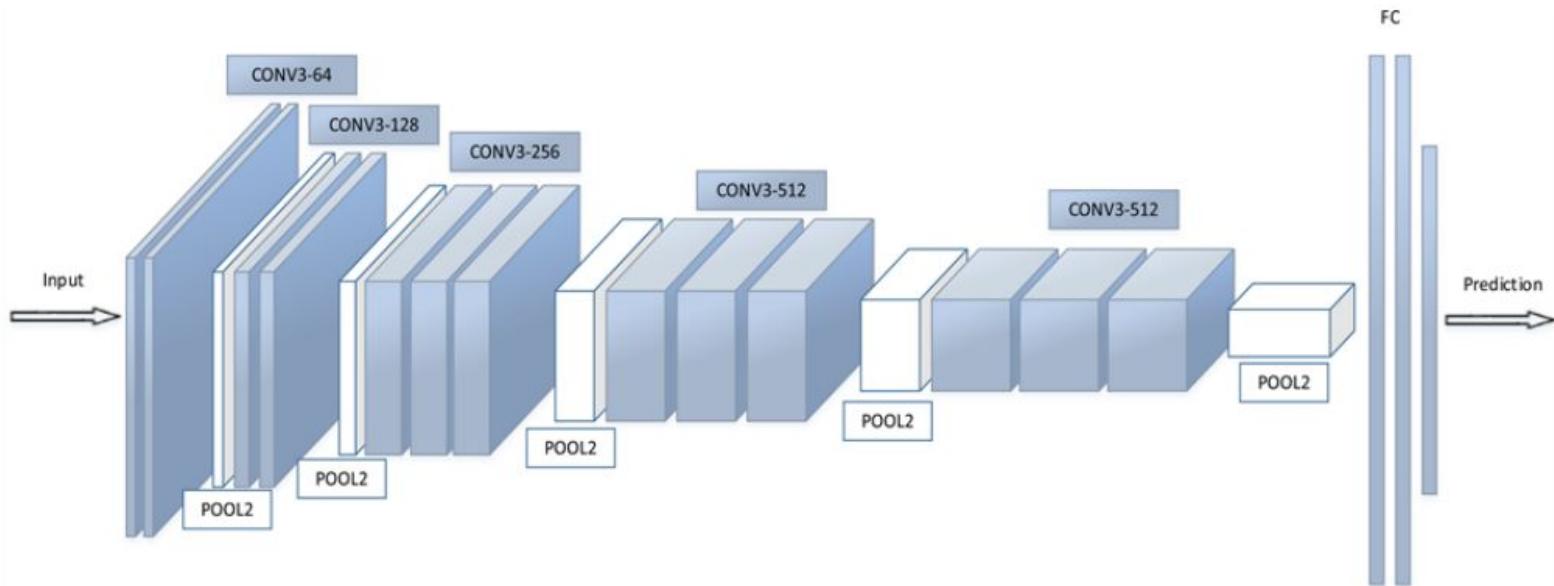
(Curso DLvis 2021 Facultad de Ingeniería UdelarR)



¿Qué es el Aprendizaje Profundo? (Deep Learning)

Clase de representaciones paramétricas no lineales capaces de codificar las características del problema y de ser optimizadas de forma eficiente usando métodos de optimización estocástica.

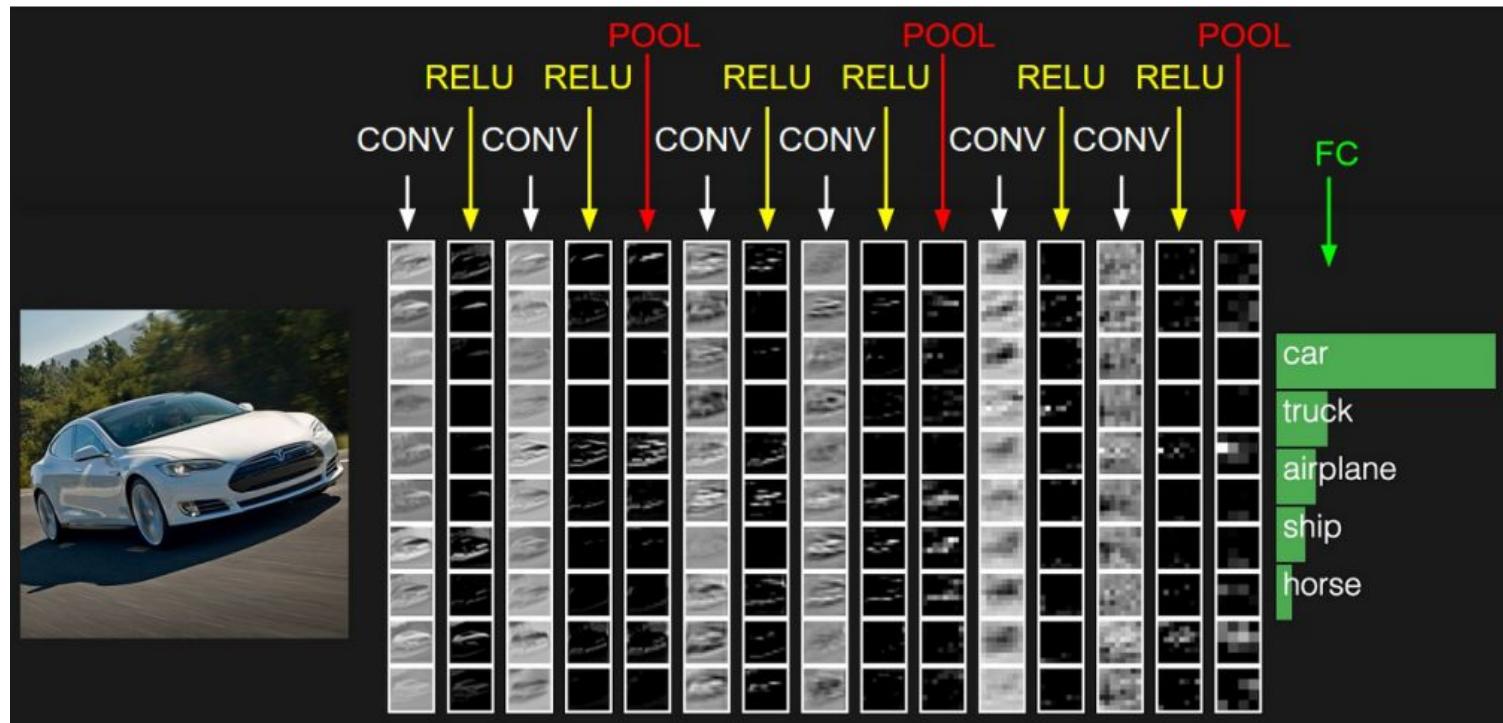
(Curso DLVIS 2021 Facultad de Ingeniería Udelar)



¿Qué es el Aprendizaje Profundo? (Deep Learning)

Clase de representaciones paramétricas no lineales capaces de codificar las características del problema y de ser optimizadas de forma eficiente usando métodos de optimización estocástica.

(Curso DLVIS 2021 Facultad de Ingeniería Udelar)

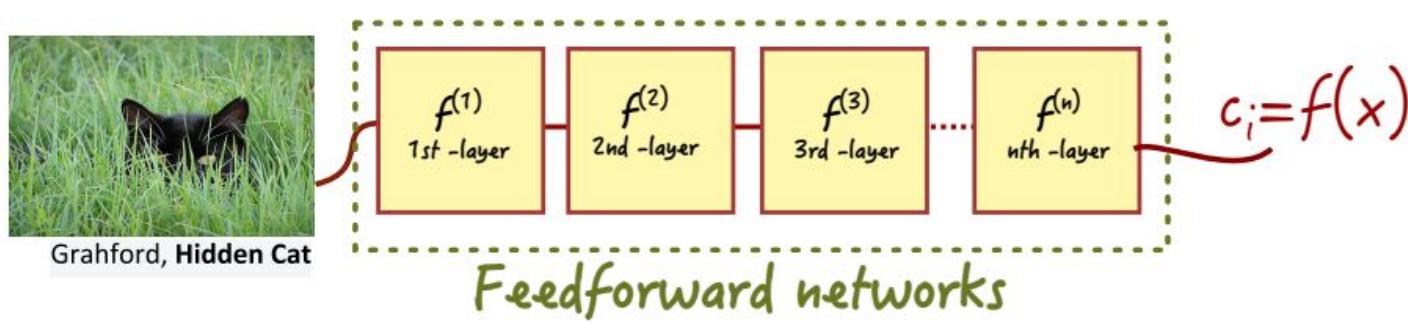


(*) Imagen de towardsdatascience

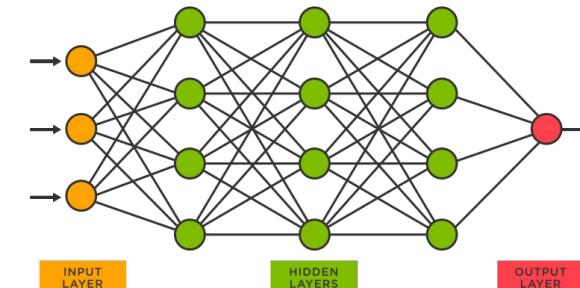
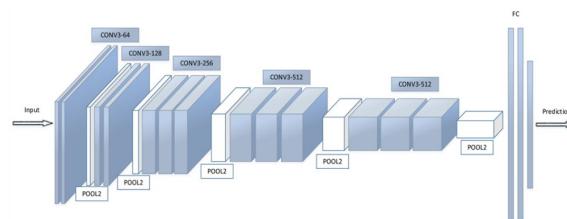


REDES NEURONALES PROFUNDAS

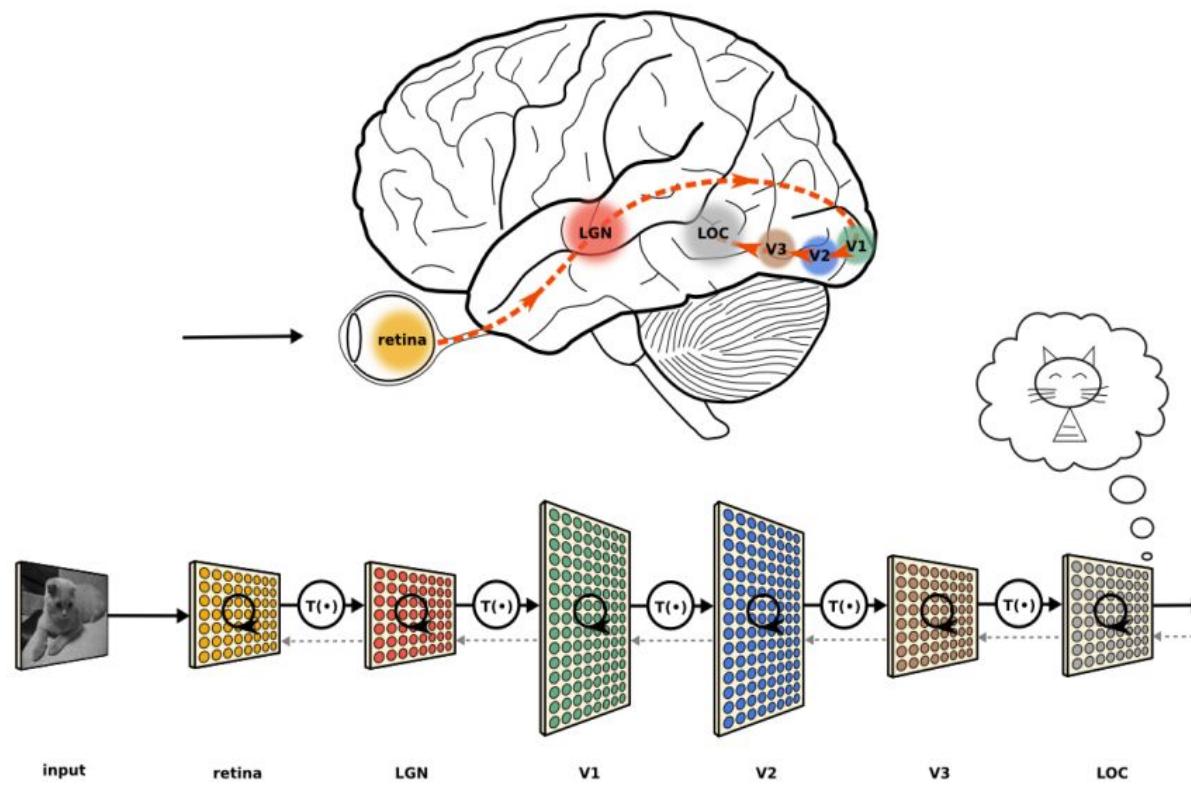
- La **profundidad (depth)** de la red: cantidad de capas en el modelo (deep)
- Primera, **capa de entrada (input layer)**, última, **capa de salida (output layer)**
- El **entrenamiento** no especifica la salida de cada capa sino de **punta a punta**.
- Datos de entrenamiento **no** es observado por las capas intermedias, estas son llamadas **capas ocultas (hidden layers)**
- Cada capa es en general una función vectorial: $f^{(j)} : \mathbb{R}^{n_j} \rightarrow \mathbb{R}^{n_{j+1}}$. Dimensión n_j determina el ancho de la capa (width).
- Cada elemento de una capa $[f^{(j)}]$, es llamado **neurona** (o unidad) : dado una serie de entradas calcula salida unidimensional (su función de activación)



- **Cascada de capas** de procesamiento no lineal para extraer y transformar variables.
- Cada capa usa la salida de la capa anterior como entrada.
- Transforma la representación anterior a otra de **mayor nivel de abstracción**.
- Múltiples niveles de representación se corresponden con **diferentes niveles de abstracción** (jerarquía de conceptos).
- Representaciones de alto nivel son **más globales y más invariantes**.
- Representaciones de bajo nivel son **comunes a las distintas categorías**.



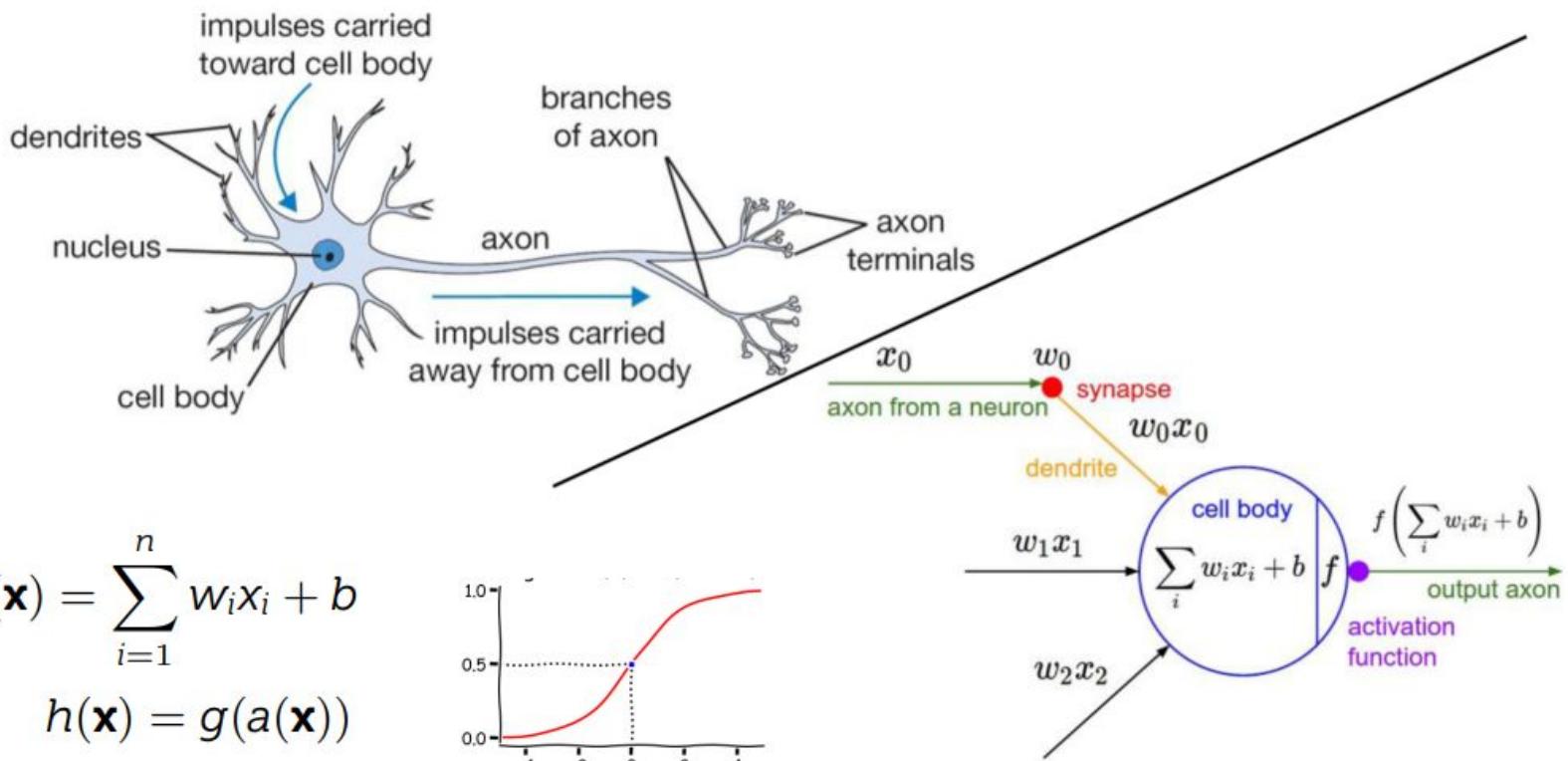
APRENDIZAJE PROFUNDO (Deep Learning)



- Representaciones intermedias: Retina - LGN - V1 - V2 - V4 - PIT - AIT
- Del orden de 100.000 millones de neuronas
- Cerebro humano no funciona como las redes de *Deep Learning*

Neuronas Artificiales

- Componente elemental del Aprendizaje Profundo
- Típicamente, está compuesta por: operación lineal + función de activación no lineal.
- El nombre viene de su “paralelismo” con las neuronas biológicas



Neuronas Artificiales

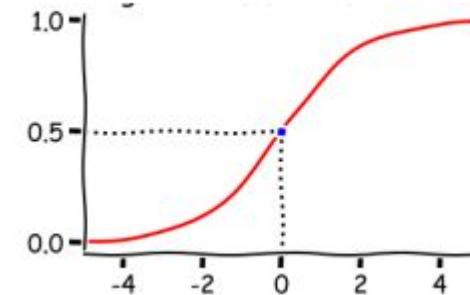
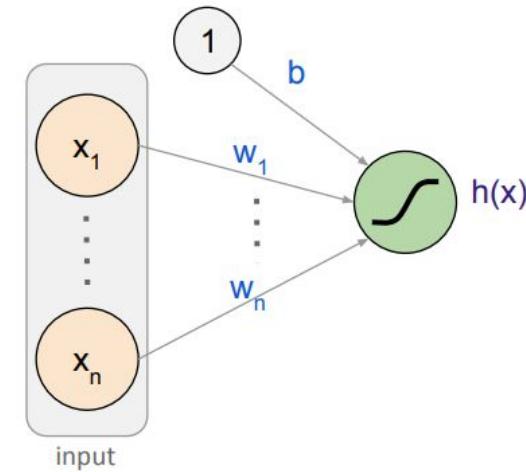
- Una neurona artificial se compone (en general) de:
una operación lineal + una función de activación no lineal

1. Pre-activación

$$a(\mathbf{x}) = \sum_{i=1}^n w_i x_i + b = \mathbf{w}^T \mathbf{x} + b$$

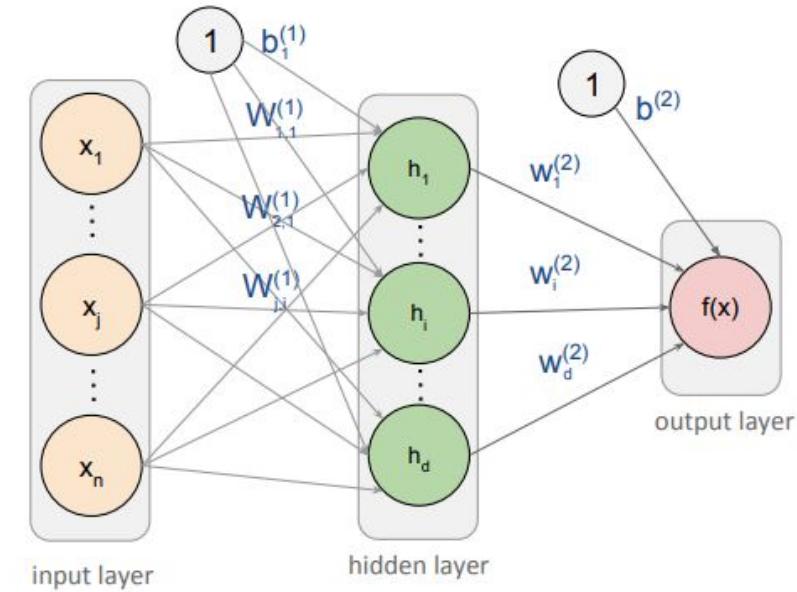
2. Activación (salida)

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g\left(\sum_{i=1}^n w_i x_i + b\right)$$



Red Neuronal de una capa oculta

- Entrada: \mathbf{x}
- Pre-activación:
$$\mathbf{a} = \mathbf{W}^{(1)\top} \mathbf{x} + \mathbf{b}^{(1)}$$
- Activación capa oculta:
$$\mathbf{h}^{(1)}(\mathbf{x}) = g(\mathbf{a}(\mathbf{x}))$$
- Capa de salida:
$$f(\mathbf{x}) = o (\mathbf{w}^{(2)\top} \mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$



Red Neuronal de múltiples capas

Red con L capas ocultas.

- Entrada:

$$\mathbf{x} (= \mathbf{h}^{(0)})$$

- Pre-activación capa (j):

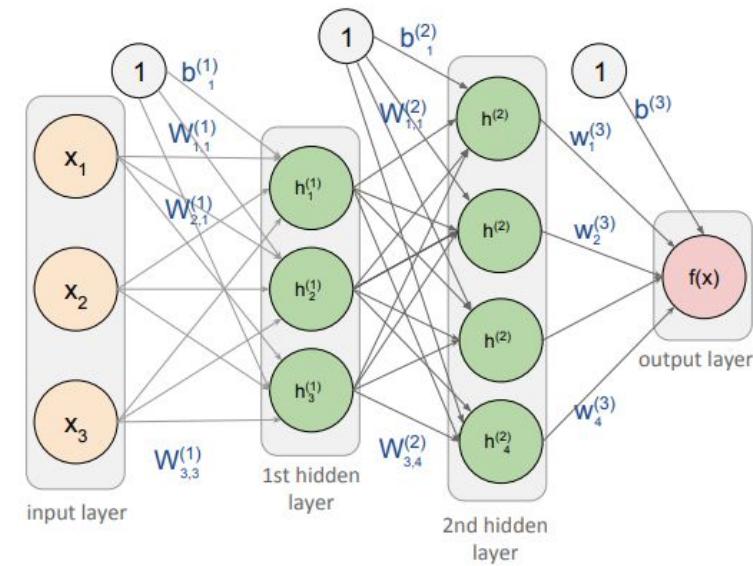
$$\mathbf{a}^{(j)} (\mathbf{x}) = \mathbf{W}^{(j)T} \mathbf{h}^{(j-1)} + \mathbf{b}^{(j)}$$

- Salida capa (j):

$$\mathbf{h}^{(j)} (\mathbf{x}) = g(\mathbf{a}^{(j)} (\mathbf{x}))$$

- Salida de la red (capa $L + 1$):

$$f(\mathbf{x}) = o (\mathbf{a}^{(L+1)})$$

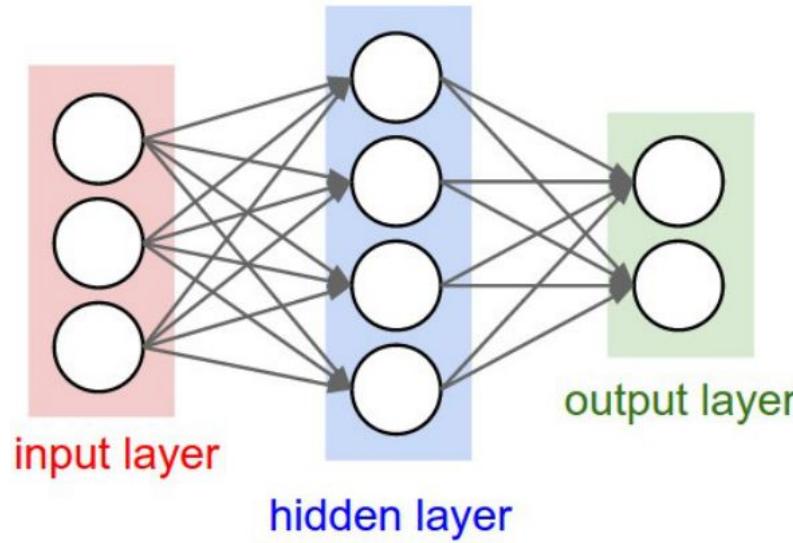


Nomenclatura:

- “3-layer neural net” o “2-hidden-layer neural net”
- Capas totalmente conectadas (“Fully-connected layers”)
- También conocido como (Multi-layer Perceptron (MLP))

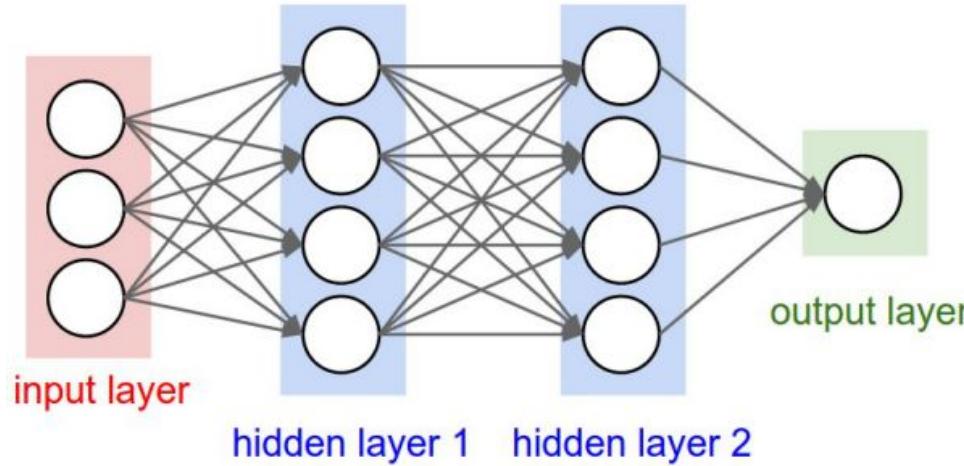


Número de parámetros de una red



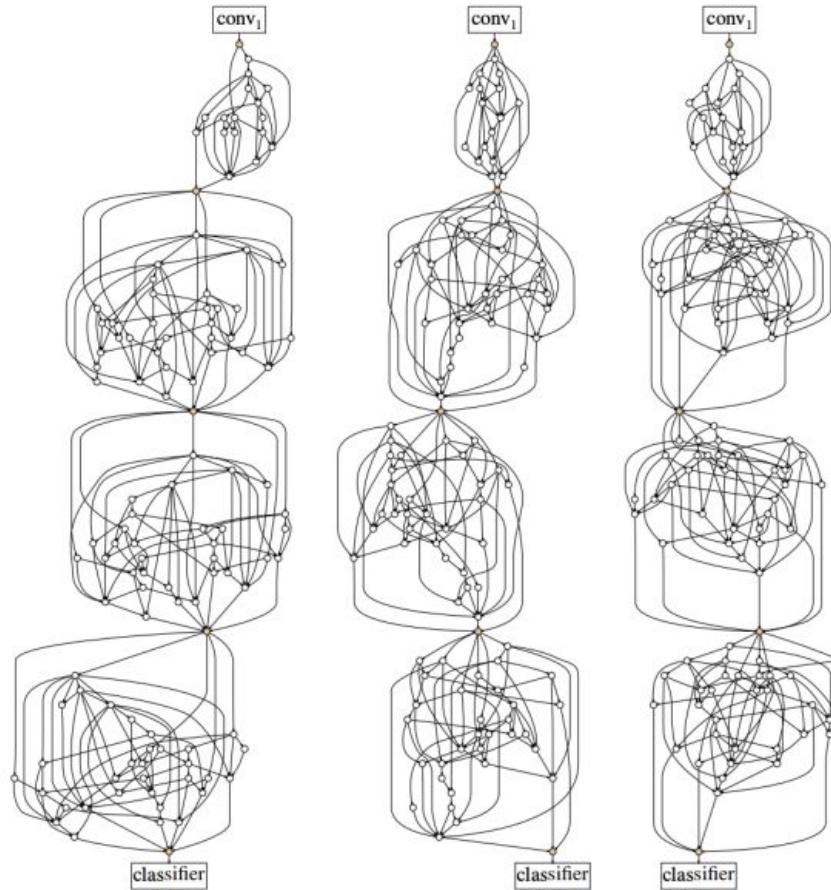
- Red 2 capas
- 1 capa oculta (4 neuronas), 1 capa de salida (2 neuronas), capa de entrada (3 entradas)
- $4 + 2 = 6$ neuronas
- $3 \times 4 + 4 \times 2 = 20$ pesos, $4 + 2$ bias = 26 parámetros

Número de parámetros de una red



- Red 3 capas
- 2 capas oculta (4 neuronas), 1 capa de salida (1 neurona), capa de entrada (3 entradas)
- $4 + 4 + 1 = 9$ neuronas
- $3 \times 4 + 4 \times 4 + 4 \times 1 = 32$ pesos, $4 + 4 + 1$ bias = $9 = 41$ parámetros

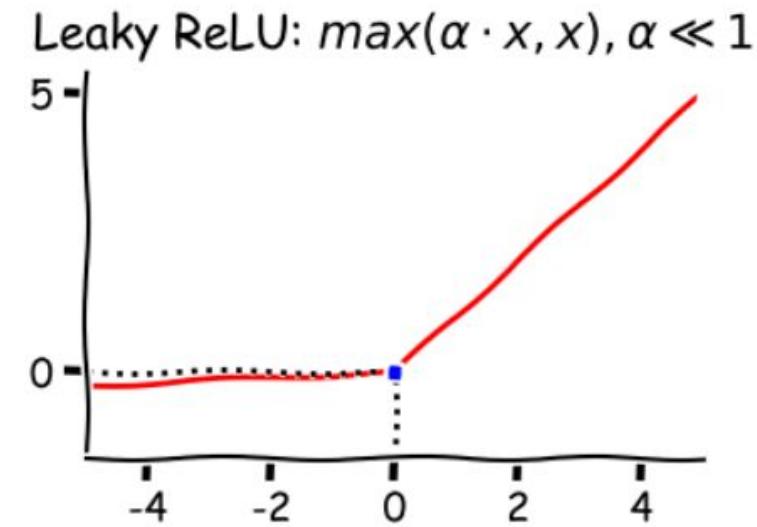
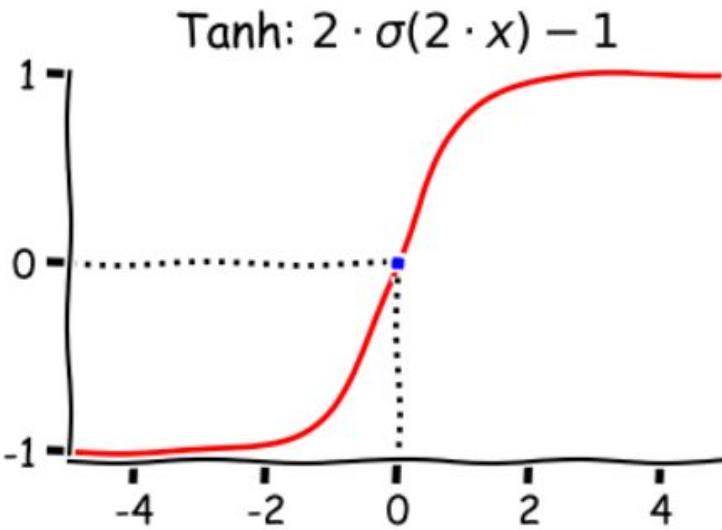
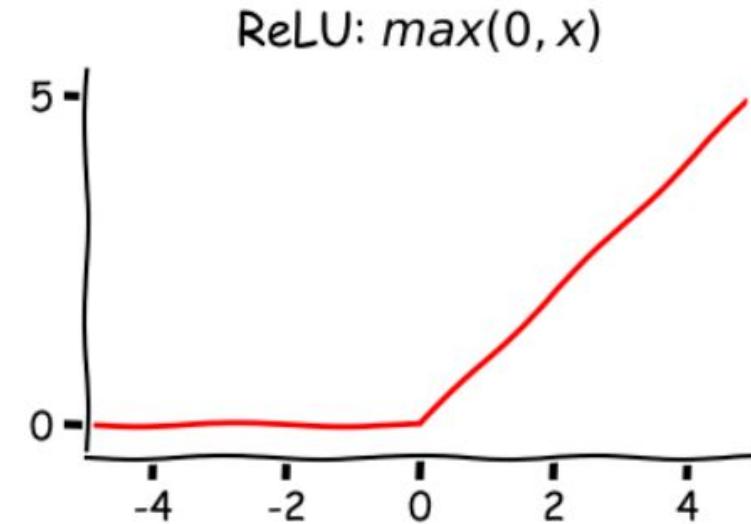
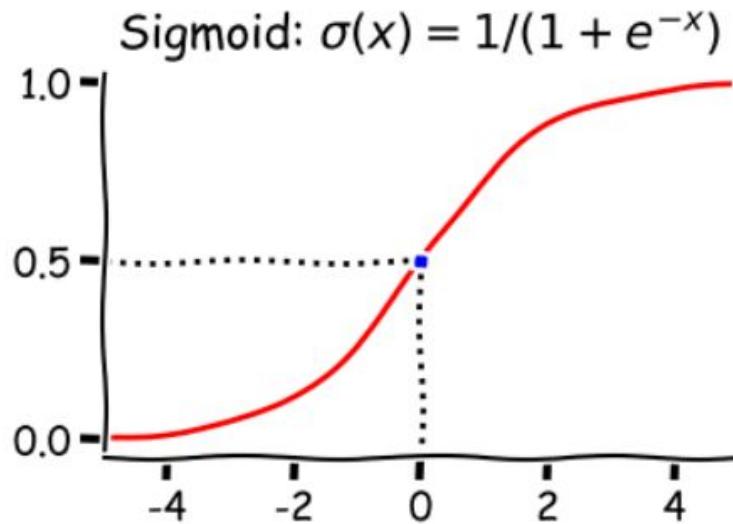
Número de parámetros de una red



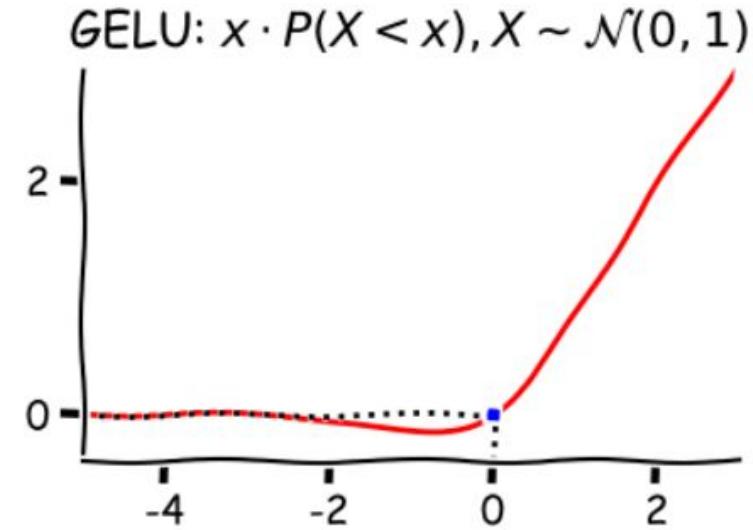
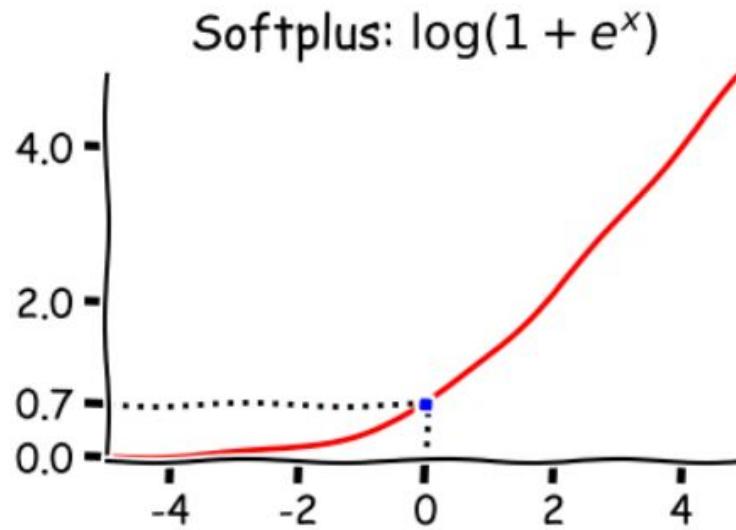
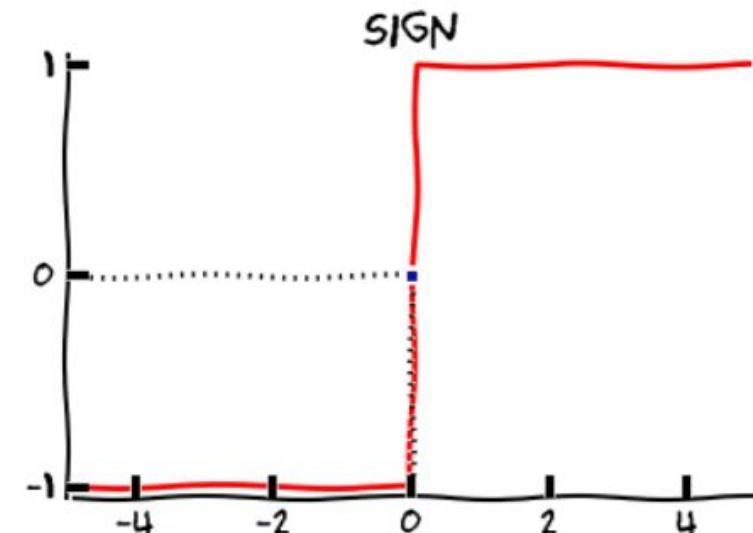
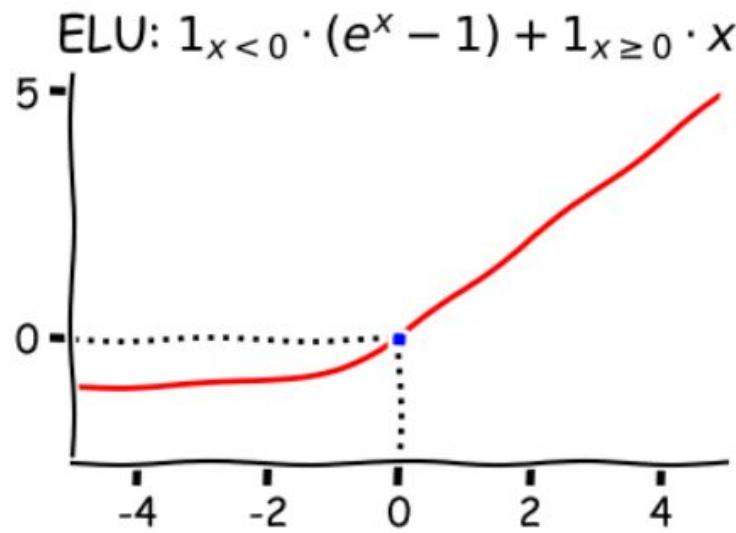
Xie et al. "Exploring Randomly Wired Neural Networks for Image Recognition", ICCV 2019



Funciones de Activación



Funciones de Activación



CAPACIDAD DE RED



- Cuantas más neuronas, más capacidad de ajustar
- **Capacidad** relacionada con **número** de neuronas ocultas y regularización

$$+ \lambda \|W\|^2 \quad \text{"weight decay"}$$

Ver demo interactiva en: <https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>

3. Teorema de aproximación universal



Teorema (Cybenko, 1989; Hornik 1991):

“Una red neuronal prealimentada con una única capa oculta y un número finito de neuronas, puede aproximar cualquier función continua en un espacio compacto de \mathbb{R}^n ”.

- Con los parámetros adecuados, se puede representar una gran variedad de funciones.
- Cybenko (1989) para función de activación sigmoide
- Hornik (1991) muestra que lo importante es la arquitectura feed-forward no función de activación.
- Leshno et al. (1993) lo demuestra para una familia más general de funciones, que incluyen a la ReLU



4. Gradient descent y Backpropagation



DESCENSO POR GRADIENTE

- Buscamos minimizar:

$$L(\mathbf{W}) = \sum_{i=1}^n L_i(\mathbf{x}_i, y_i; \mathbf{W}) + \lambda R(\mathbf{W})$$

- **Descenso por gradiente:**

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \nabla_{\mathbf{W}} L(\mathbf{W}_t) \quad \eta > 0 \text{ ("learning rate").}$$

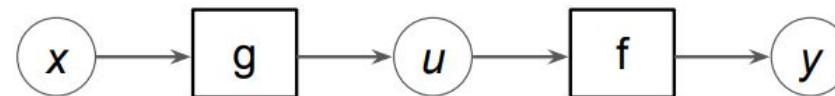
- ¿Cómo calculamos analíticamente $\nabla_{\mathbf{W}} L(\mathbf{W})$?
- Es decir, ¿cuál es la influencia de cada parámetro en la función de costo?
- El cálculo analítico es engorroso y poco flexible a modificaciones en la arquitectura del modelo.
- **El método de backpropagation ofrece una forma práctica de calcular este gradiente.**



EJEMPLO

Objetivo: determinar cuánto varía el valor de una función al variar una variable (o parámetro) entorno a cierto punto.

Ejemplo:



$$y = f(u) = f(g(x))$$

$$dy = du \cdot \left(\frac{dy}{du} \right) = du \cdot f'(u)$$

$$du = dx \cdot \left(\frac{du}{dx} \right) = dx \cdot g'(x)$$

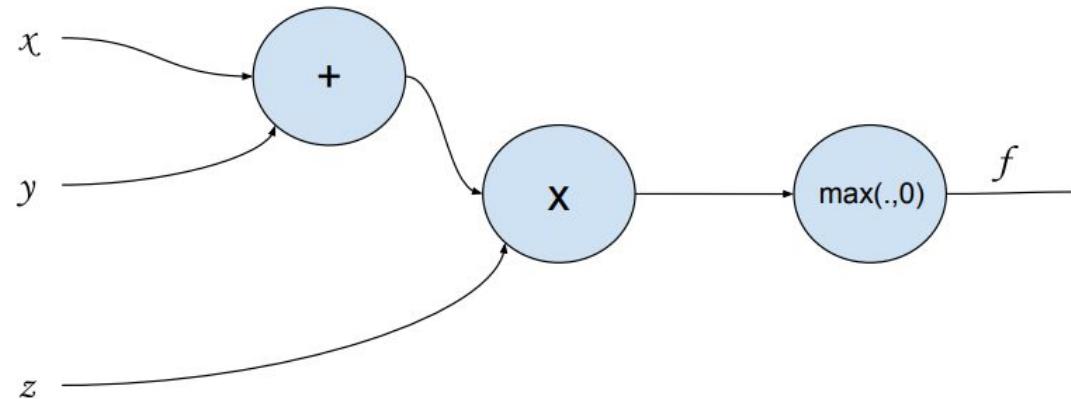
$$dy = dx \cdot \left(\frac{du}{dx} \right) \left(\frac{dy}{du} \right)$$

Regla de la cadena

Método de Backpropagation

- El método de Backpropagation es una forma de aplicar sistemáticamente la **regla de la cadena**
- Se basa en tres ideas:
 1. Representar las operaciones de la red en un **Grafo Computacional**
 2. Aplicar el paradigma **Divide-and-conquer**
 - a. Subdividir el problema principal en subpartes que se puedan resolver eficientemente
 - b. La combinación de las sub-soluciones también debe ser eficiente
 3. Evitar el cálculo explícito de matrices Jacobianas de gran tamaño

$$f(x, y, z) = \max((x + y)z, 0)$$



Grafo Computacional: Ejemplo

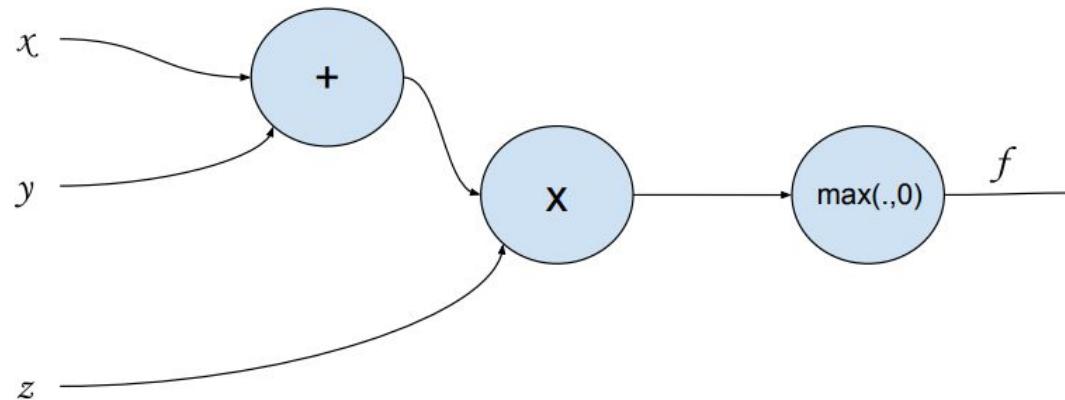
$$f(x, y, z) = \max((x + y)z, 0)$$

Composición de funciones elementales:

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

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

$$f_{\text{max}}(x) = \max(x, 0) \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1_{\{x>0\}}$$



Grafo Computacional: Ejemplo

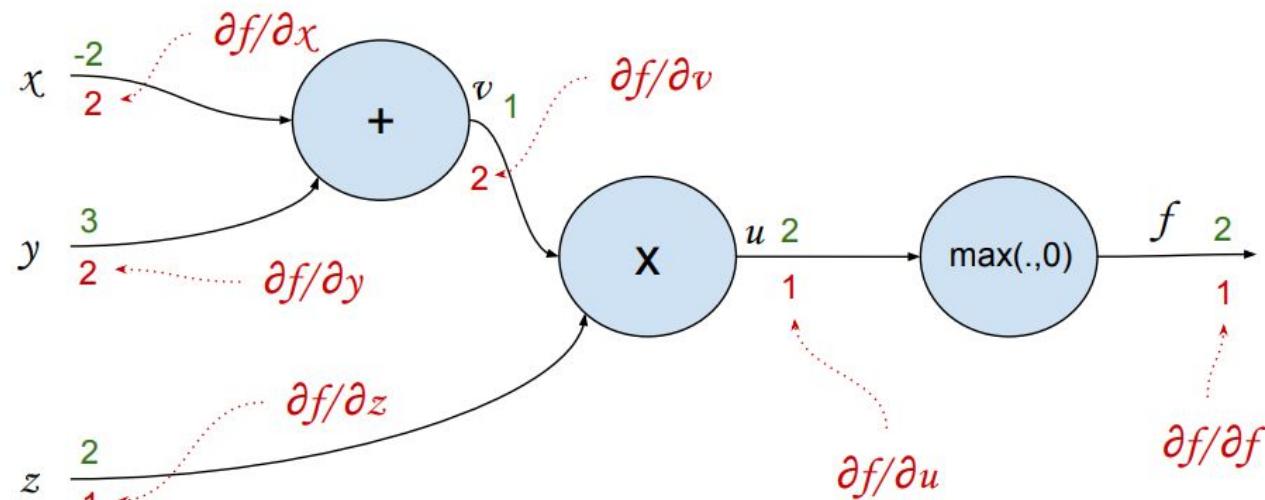
$$f(x, y, z) = \max((x + y)z, 0)$$

Composición de funciones elementales:

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

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

$$f_{\text{max}}(x) = \max(x, 0) \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1_{\{x>0\}}$$



Backpropagation: Resumen

- **Objetivo:** Evaluar cuánto varía la función de costo al variar cada parámetro de la red.
- Aplicación sistemática de la regla de la cadena de la derivada
- Representación de la red en un “Grafo Computacional”
- **Divide-and-Conquer:** Cada nodo del grafo calcula localmente sus derivadas “hacia atrás” a partir de la derivada “hacia adelante” y la derivada “local”
- El algoritmo de *backpropagation* calcula secuencialmente la derivada hacia atrás de cada nodo, empezando desde (la salida) hacia atrás (la entrada)
- Se busca además evitar el cálculo explícito de las matrices Jacobianas de gran tamaño

Para entender algoritmo de Backpropagation leer los siguientes:

- Intuitive understanding of Backpropagation cs231n Stanford
<https://cs231n.github.io/optimization-2/>
- cs231n Lecture 4: Backpropagation and Neural Networks
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf



5. Algoritmos de optimización y regularización



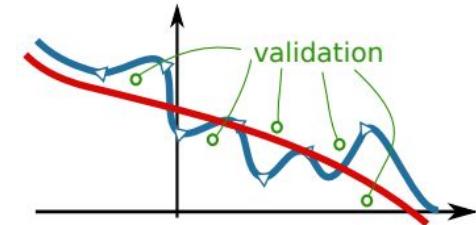
REGULARIZACIÓN

$$L(W) = \frac{1}{n} \sum_{i=1}^n L_i(f(\mathbf{x}_i, W), y_i) + \lambda R(W)$$

λ peso de regularización (hiperparámetro)

Objetivo: Encontrar W que ajuste bien datos de entrenamiento y que generalice bien a datos no conocidos.

- Modelo de Compromiso:
Sobreajuste - Subajuste
- Agregar Regularización: $+ \lambda R(W)$
Ejemplo: $R(W) = \sum_{k,l} W_{k,l}^2$ (weight decay)



Occam's Razor

“Entre todas las hipótesis la más simple es la mejor”

TIPOS DE REGULARIZACIÓN

$$L(W) = \frac{1}{n} \sum_{i=1}^n L_i(f(\mathbf{x}_i, W), y_i) + \lambda R(W)$$

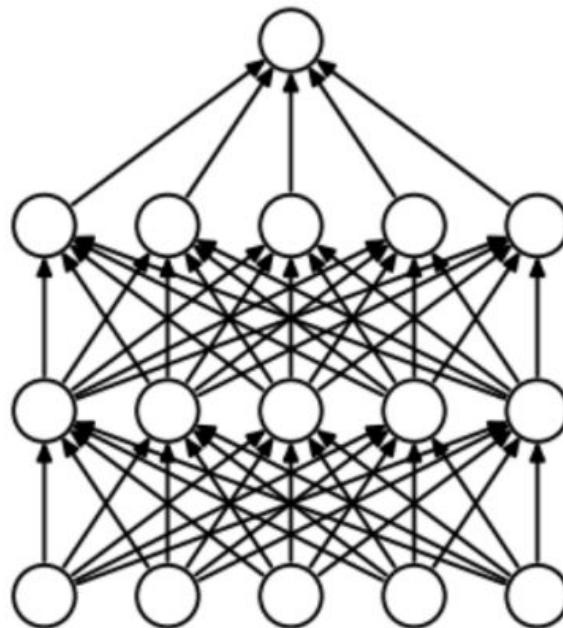
λ peso de regularización (hiperparámetro)

- Penalización $L2$: $R(W) = \|W\|^2 = \sum_{k,l} W_{k,l}^2$
- Penalización $L1$: $R(W) = \|W\|_1 = \sum_{k,l} |W_{k,l}|$
- $L2 + L1$ (elastic net): $R(W) = \beta \|W\|^2 + \|W\|_1$
- Pesos no negativos $W_{k,l} \geq 0$
- Compartir pesos (e.g., estructura dada en W)
- *Dropout*
- *Batch normalization*

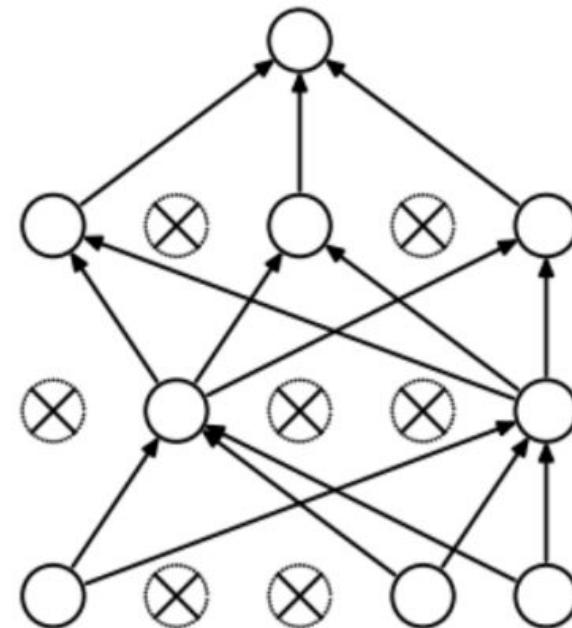


REGULARIZACIÓN: DROPOUT

En la pasada hacia adelante durante el entrenamiento, se fijan a cero algunas neuronas de manera aleatoria (con probabilidad p).



(a) Standard Neural Net



(b) After applying dropout.

Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I. and Salakhutdinov,
“Improving neural networks by preventing co-adaptation of feature detectors.” arXiv 2012

REGULARIZACIÓN: DROPOUT

Para no enlentecer la etapa de testing, se hace **dropout invertido** (re-escalado durante el entrenamiento). El código de predicción es igual que si no se hace dropout.

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

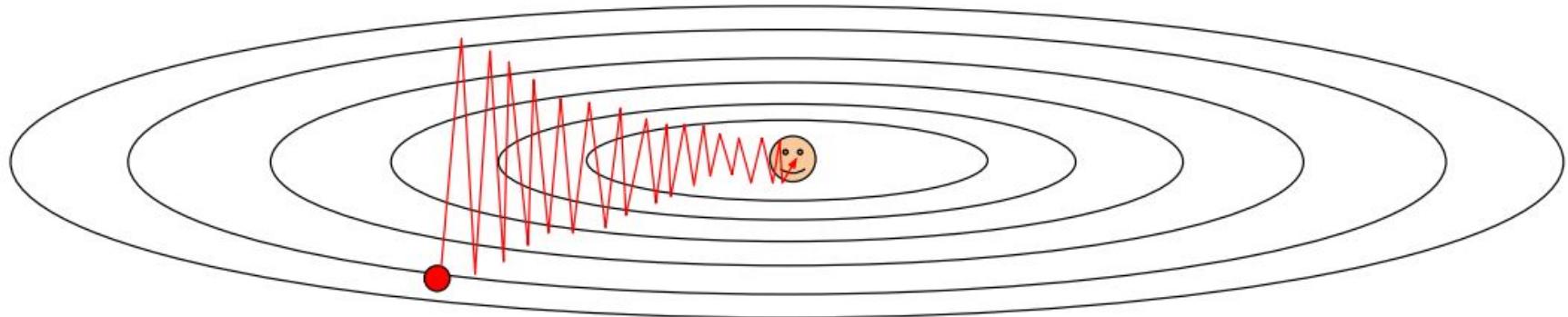
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```



Descenso por gradiente: desafíos

¿Qué sucede si la loss es mal condicionada?

(mal condicionada: cambia mucho en una dirección y poco en otra)



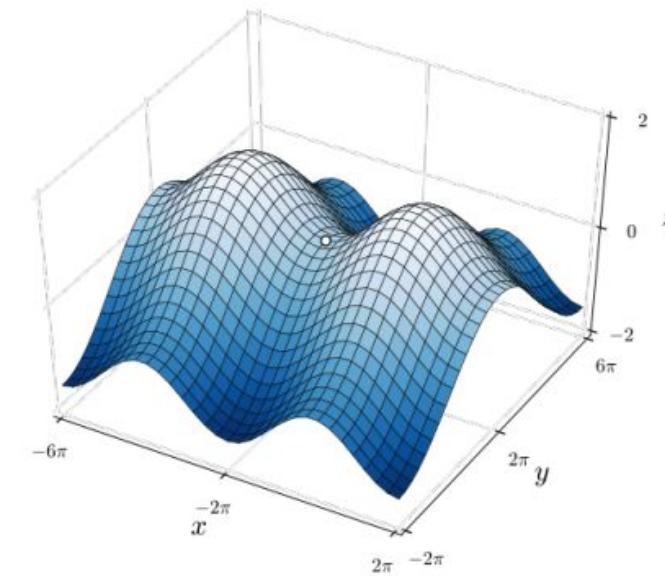
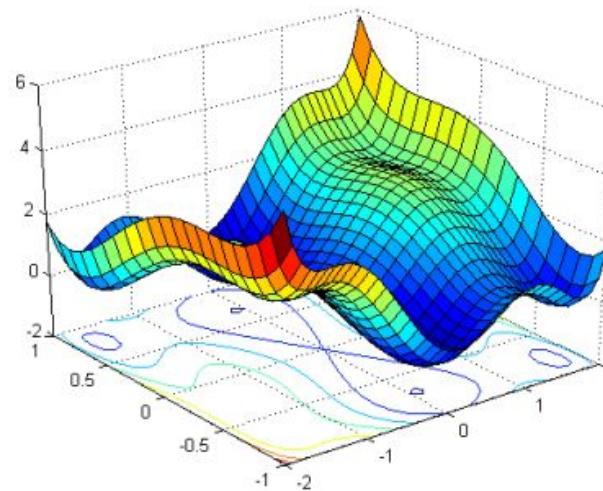
Mal condicionamiento se mide por el número de condición del Hessiano \mathbf{H} (derivada segunda de la loss):

- Demora mucho en converger...
- Se avanza muy lento en la dirección de valor propio pequeño;
- Se oscila en la dirección de más alto valor propio

Descenso por gradiente: desafíos

¿Qué sucede con los mínimos locales y los puntos silla?

- En **mínimos locales o puntos silla**: $\nabla L(W) = 0$; GD se detiene.
- Punto silla: Hessiano **H** tiene valores propios de distinto signo; en alta dimensión es frecuente
- Cerca de los puntos silla: se avanza muy muy lento



1. **SGD (Stochastic Gradient Descent)**: Actualizar weights con Loss para todos los datos del dataset puede tomar mucho tiempo con datasets muy grandes. Solución: aproximar con mini batches.
2. **Momentum SGD**: Diseñado para tener inercia y converger más rápido
3. **Nesterov Momentum**: Similar al anterior pero con mayor inteligencia al momento de calcular la inercia.
4. **AdaGrad**: Re-escalado en cada dimensión inversamente proporcional a la suma histórica de los cuadrados del gradiente.
5. **RMSProp**: Se mantiene una media móvil con el cuadrado del gradiente para cada componente.
6. **ADAM**: Mantener inercia en cada componente. Ir “frenando” cada componente, manteniendo un estimador del cuadrado del gradiente y dividiendo el learning rate



Momentum SGD

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

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

SGD

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

- Diseñado para tener **inercia**
- ρ : parámetro de memoria/fricción; típicamente $\rho = 0.9$ o 0.99
- **Dirección de descenso**: promedio de velocidad y gradiente
- Especialmente adaptado para cuando hay alta curvatura o ruido en el gradiente
- v : velocidad, $v_0 = 0$

<https://distill.pub/2017/momentum/>



Nesterov Momentum

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

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

Momentum SGD

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

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

- **Nesterov:** Calcular el gradiente en el punto que se llegaría si se continúa a la misma velocidad v .
- En caso convexo se puede probar que acelera la convergencia.

Nesterov, Y.

“A method of solving a convex programming problem with convergence rate $O(1/k^2)$ ”, 1983

Sutskever, I., Martens, J., Dahl, G. and Hinton, G.

“On the importance of initialization and momentum in deep learning”, ICML, 2013



1. AdaGrad: Re-escalado en cada dimensión inversamente proporcional a la suma histórica de los cuadrados del gradiente.

Duchi, John, Elad Hazan, and Yoram Singer "Adaptive subgradient methods for online learning and stochastic optimization." JMLR, 2011

2. RMSProp: Se mantiene una media móvil con el cuadrado del gradiente para cada componente.

T. Tieleman, G. Hinton. Coursera: Neural networks for machine learning, 2012. "Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude."

3. ADAM (Adaptative Moments): Mantener inercia e ir “frenando” cada componente, manteniendo un estimador del cuadrado del gradiente y dividiendo el learning rate.

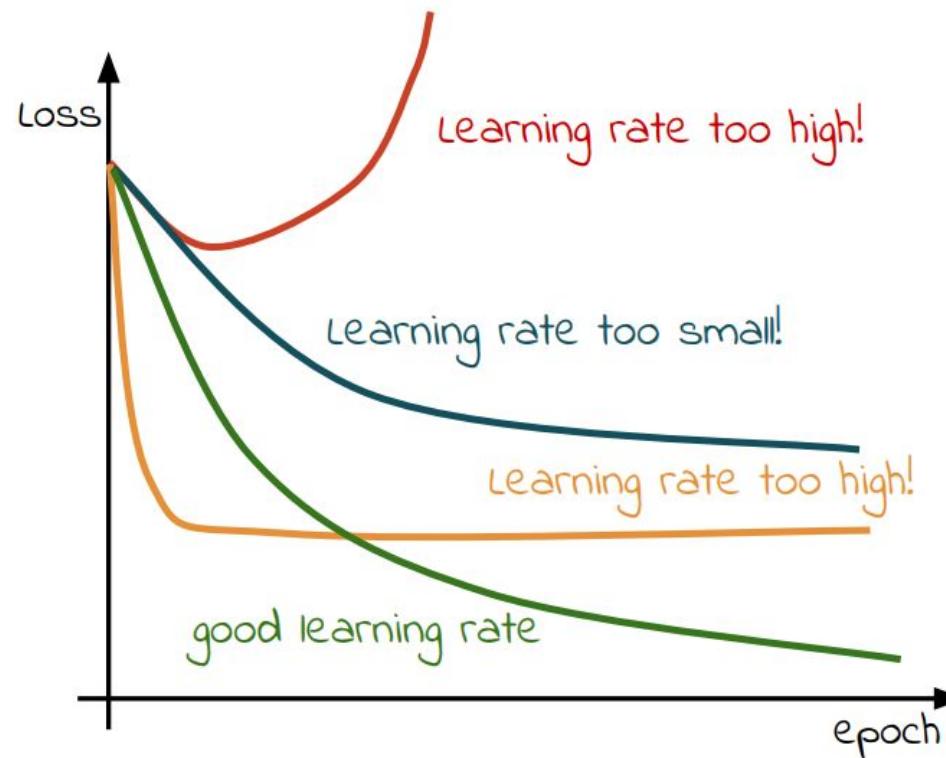
Kingma, Diederik, and Jimmy Ba. "Adam: A method for stochastic optimization.", ICLR 2015



Learning Rate

Todos los métodos vistos tienen un hiperparámetro: learning rate

¿Cómo elegimos el learning rate?



6. Herramientas y bibliotecas



Librerías para Aprendizaje Profundo

- Caffe → Caffe2 (UC Berkeley → Facebook)
- Theano → TensorFlow (U. Montreal → Google)
- Torch → PyTorch (NYU → Facebook)
- De alto nivel: Keras (sobre TensorFlow), Sonnet (DeepMind sobre TensorFlow), fastai (PyTorch)
- Otras: MatConvNet (Oxford Univ (VGG)), MXNet (Amazon), CNTK (Microsoft), Paddle (Baidu), Trax (Google Brain), etc.
- Proyectos de unificación: Open Neural NEtwork Exchange (ONNX)



Keras



- Biblioteca propuesta originalmente por Google en “*TensorFlow: Large-Scale Machine LEarning on Heterogeneous Distributed Systems, 2015*”
- Versiones 1.x y 2.x completamente diferentes
- Soporte a grafo computacional dinámico por defecto (opcional estático)

Ventajas:

- ✓ Python + Numpy
- ✓ Muy usado, gran comunidad y respaldo de Google
- ✓ Versión LITE y otras herramientas de la suite
- ✓ Universalidad: CPU, GPU, TPU
- ✓ Grafos dinámicos en v2.x



Tensor Flow

1. Crea los tensores para los pesos.

Notar que se us **tf.Variable** para los pesos.

2. Utilizando scope de Python y **GradientTape** se le indica a TensorFlow que construya el grafo computacional.

3. Calcular gradientes.

4. Paso descenso por gradiente:
actualizar pesos.

```
import tensorflow as tf

N, Din, H, Dout = 16, 1000, 100, 10

x = tf.random.normal((N, Din))
y = tf.random.normal((N, Dout))
w1 = tf.Variable(tf.random.normal((Din, H)))
w2 = tf.Variable(tf.random.normal((H, Dout)))

learning_rate = 1e-6
for t in range(1000):
    with tf.GradientTape() as tape:
        h = tf.maximum(tf.matmul(x, w1), 0)
        y_pred = tf.matmul(h, w2)
        diff = y_pred - y
        loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

    grad_w1, grad_w2 = tape.gradient(loss, [w1, w2])

    w1.assign(w1 - learning_rate * grad_w1)
    w2.assign(w2 - learning_rate * grad_w2)
```



REFERENCIAS

- CS231n: Deep Learning for Computer Vision - Stanford
<http://cs231n.stanford.edu/>
- CS231n: Optimization: Stochastic Gradient Descent
<https://cs231n.github.io/optimization-1/>
- CS231n: Backpropagation, Intuitions
<https://cs231n.github.io/optimization-2/>
- CS231n: Neural Networks Part 1: Setting up the Architecture
<https://cs231n.github.io/neural-networks-1/>
- CS231n: Neural Networks Part 2: Setting up the Data and the Loss
<https://cs231n.github.io/neural-networks-2/>
- CS231n: Neural Networks Part 3: Learning and Evaluation
<https://cs231n.github.io/neural-networks-3/>
- Backpropagation for a linear layer
https://eva.fing.edu.uy/pluginfile.php/178739/mod_resource/content/2/linear-bac_kprop.pdf



REFERENCIAS

- Fing - Aprendizaje Profundo para Visión Artificial (DLvis)
<https://eva.fing.edu.uy/course/view.php?id=1046>
- Fing - DLvis: Clase 4 - Introducción Redes Neuronales
https://eva.fing.edu.uy/pluginfile.php/315603/mod_resource/content/2/c4.pdf
- Fing - DLvis: Clase 5 - Algoritmo Backpropagation
https://eva.fing.edu.uy/pluginfile.php/315960/mod_resource/content/2/c5.pdf
- Fing - DLvis: Clase 7 - Regularización
https://eva.fing.edu.uy/pluginfile.php/316644/mod_resource/content/2/c7.pdf
- Fing - DLvis: Clase 8 SGD, Momentum, ADAM
https://iee.fing.edu.uy/~pmuse/DLVIS2020/DLVIS2020_clase8.pdf
- Fing - DLvis: Clase 10 - Librerías Deep Learning
https://eva.fing.edu.uy/pluginfile.php/317637/mod_resource/content/3/c10_light.pdf

