

Pontificia Universidad Católica de Chile  
Escuela de Ingeniería  
Departamento de Ciencia de la Computación



# IIC2613 - Inteligencia Artificial

Entrenamiento de Redes Neuronales

Hans Löbel  
Dpto. Ingeniería de Transporte y Logística  
Dpto. Ciencia de la Computación

En teoría, no hay diferencia entre la teoría y la práctica. En la práctica, sí la hay.

Tras bambalinas, para entrenar los modelos de DL adecuadamente se requiere una gran cantidad de detalles:

- ¿Cómo plantear y resolver el problema de aprendizaje de una red neuronal profunda?
- ¿Cómo controlar el sobreajuste en el entrenamiento?

Recordemos un poco cómo se ve un problema de aprendizaje en ML desde el punto de vista de la optimización

$$\operatorname{argmin}_W J(X, Y; W) = \lambda \mathcal{R}(W) + \sum_i^N \mathcal{L}(x_i, y_i; W)$$



Regularización



Suma de la pérdida por  
cada ejemplo

## Comencemos con un simple Perceptrón y su entrenamiento de manera supervisada

- Si tenemos suficientes datos, pares  $(x_i, y_i)$ , buscamos construir una función de pérdida, que nos indique cuán buena es en promedio la estimación del Perceptrón.
- Sea  $f(x; w) = \sigma(\sum_{i=1}^m w_{(i)}x_{(i)}) = \sigma(w \cdot x)$ , una función que modela la aplicación de un Perceptrón lineal\* parametrizado por un vector de pesos  $w$ , a un vector de características  $x$ .
- Sin suponer cosas raras sobre los datos, una posible función de pérdida (o costo, o error) puede quedar de la siguiente manera:

$$J(w) = \frac{1}{2n} \sum_{i=1}^n (f(x_i; w) - y_i)^2$$

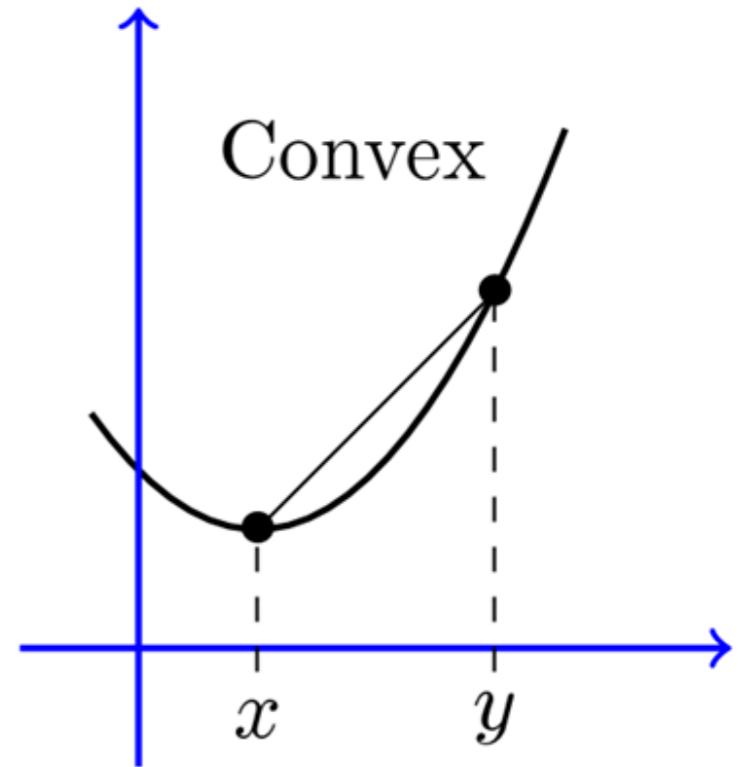
- ¿Qué representa el valor mínimo de esta función, con respecto a los pesos  $w$ ?

\* función identidad como activación

¿Cómo podemos entrenar un Perceptron (lineal) para estimar funciones?

- La función de pérdida es convexa (siempre que la función de activación también lo sea, lo que ocurre regularmente)

$$J(w) = \frac{1}{2n} \sum_{i=1}^n (f(x_i; w) - y_i)^2$$



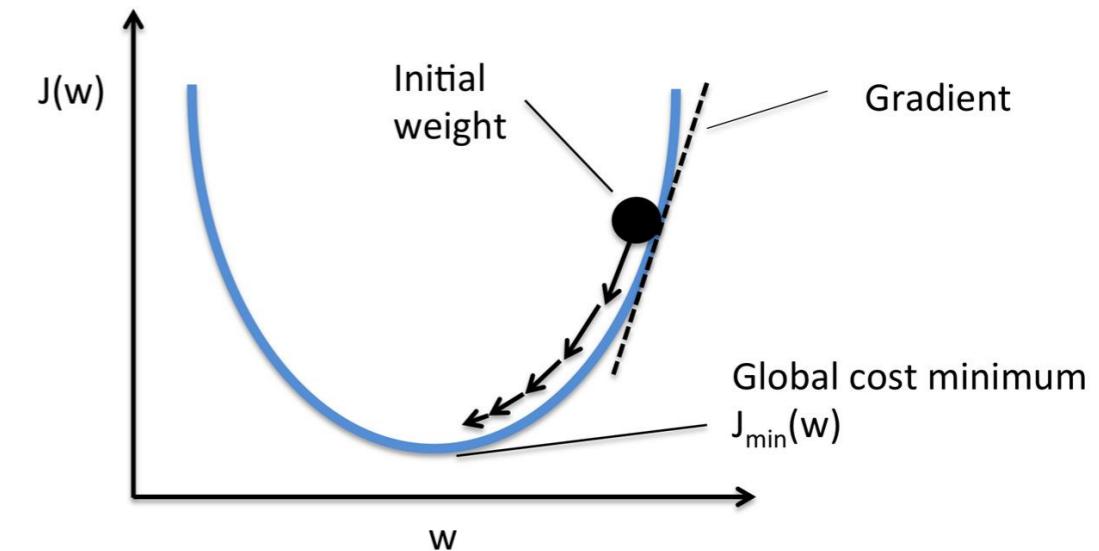
# ¿Cómo podemos entrenar un Perceptron (lineal) para estimar funciones?

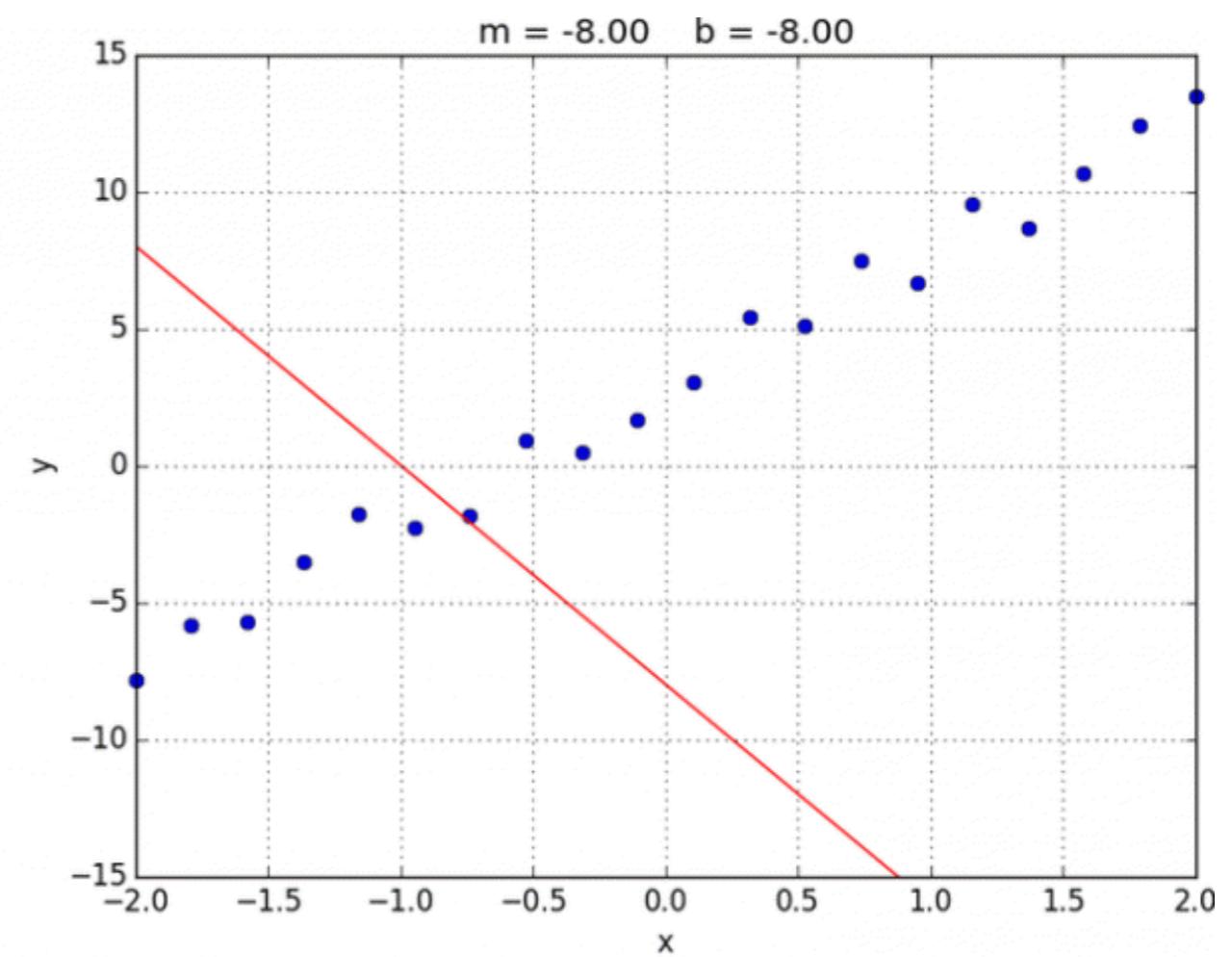
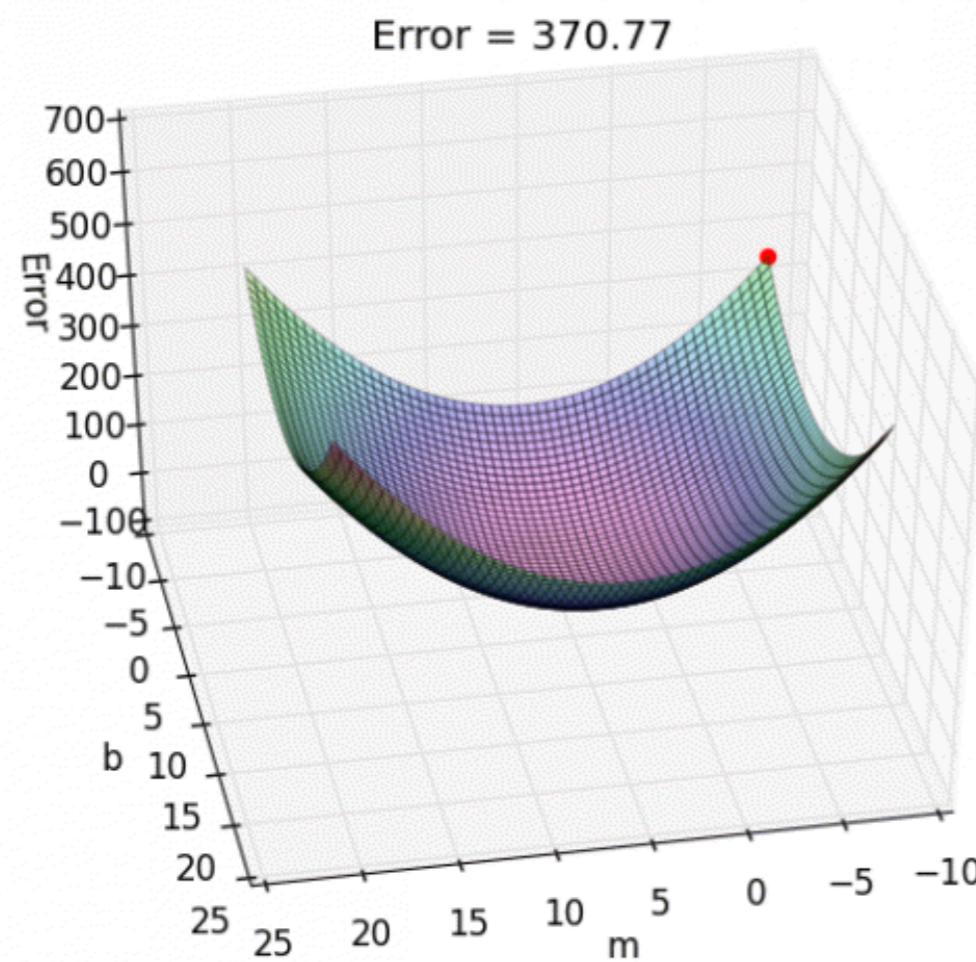
- La función de pérdida es convexa (si la función de activación también lo es, lo que ocurre regularmente)

$$J(w) = \frac{1}{2n} \sum_{i=1}^n (f(x_i; w) - y_i)^2$$

- Dado esto, es posible utilizar el algoritmo de descenso de gradiente

btw, ¿cuál es la derivada de la función de pérdida con respecto a  $w$ ?





Dado que el problema de aprendizaje es convexo, podemos usar “tranquilamente” descenso de gradiente

---

**Require:** Pesos  $w_{(i)}$  inicializados con un valor aleatorio pequeño  
**do**

$$\Delta w \leftarrow \vec{0}$$

**for each**  $(x, y) \in \mathcal{S}$  **do**

Aplicar Perceptrón a  $x$  y almacenar salida en  $\hat{y}$

**for each**  $\Delta w_{(i)} \in \Delta w$  **do**

$$\Delta w_{(i)} \leftarrow \Delta w_{(i)} + (y - \hat{y})x_{(i)}$$

**end for**

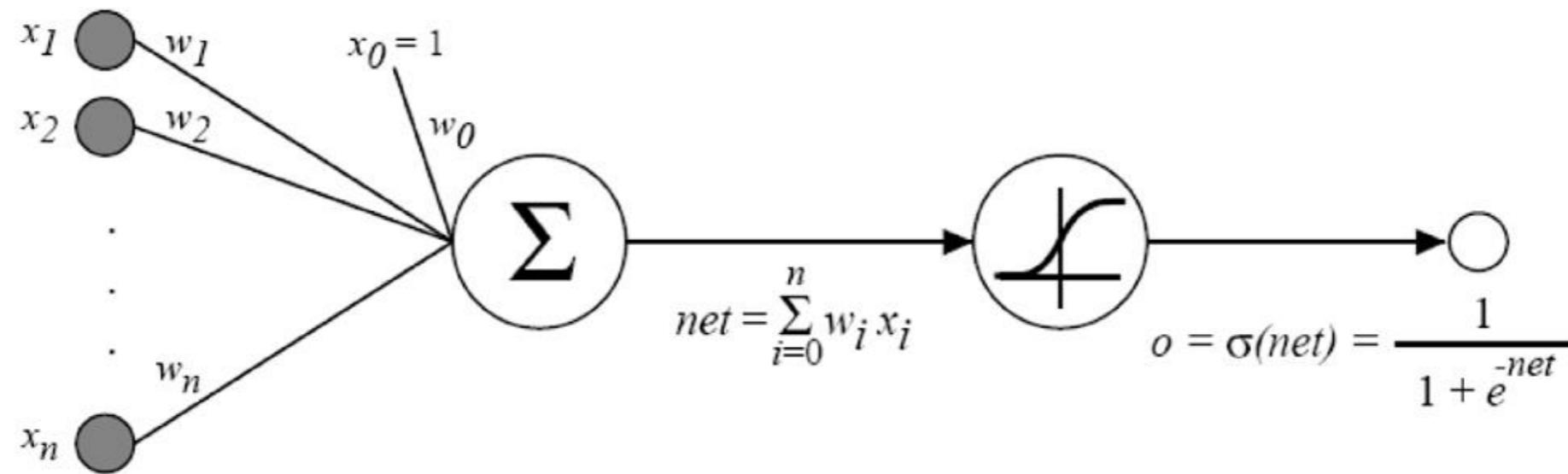
**end for**

$$w \leftarrow w - \alpha \frac{\Delta w}{|\mathcal{S}|}$$

**while** condición de término == **False**

---

Si incorporamos una función de activación no lineal, la situación no cambia mucho



$$\frac{\partial J}{\partial w_i} = - \sum_{d \in D} (t_d - o_d)o_d(1 - o_d)x_{i,d}$$

Movámonos ahora a un MLP, que rápidamente expondrá las dificultades al entrenar redes neuronales

Siempre es bueno pensar también en las redes neuronales\* como una composición de funciones:

- MLP de 1 capa  
(Perceptron):

$$f(x) = \sigma(W_1 x)$$

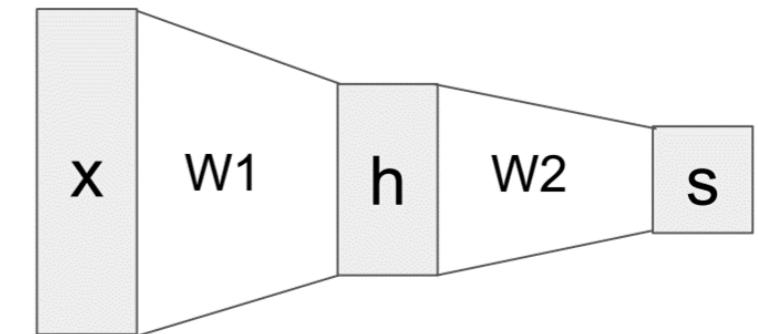
\*  $\sigma(z)$  = función de activación

- MLP de 2 capas:

$$f(x) = \sigma(W_2 \sigma(W_1 x))$$

- MLP de 3 capas:

$$f(x) = \sigma(W_3 \sigma(W_2 \sigma(W_1 x)))$$



\* Con un poco de masaje algebraico, todo esto es también válido para una CNN, o cualquier otra red

## ¿Cómo queda entonces el problema de aprendizaje de un MLP?

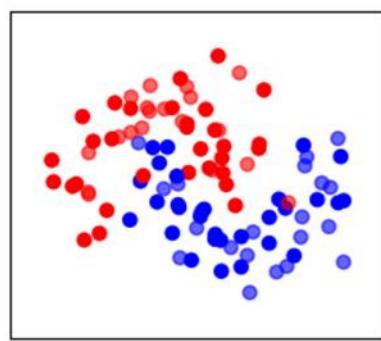
- Al igual que un Perceptron, podemos definir fácilmente una función de pérdida:

$$J(W) = \frac{1}{2n} \sum_{i=1}^n (f(x_i; W) - y_i)^2$$

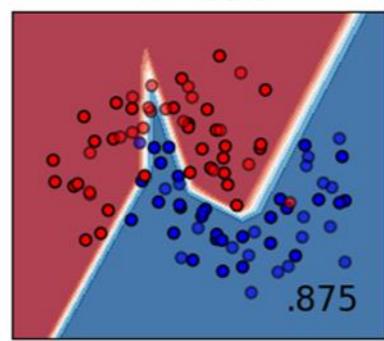
- Sin embargo, acá debemos considerar 3 nuevos elementos:
  1. Ahora tenemos muchos más parámetros que antes, ¿cómo puedo controlar la complejidad?

Podemos tomar prestados conceptos de los SVM:

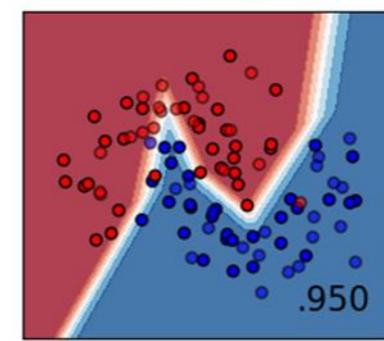
$$J(W) = \frac{C}{2} \|W\|^2 + \frac{1}{2n} \sum_{i=1}^n (f(x_i; W) - y_i)^2$$



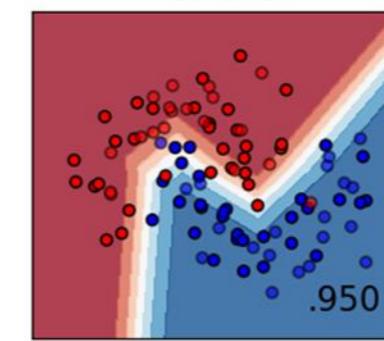
$C = 0.10$



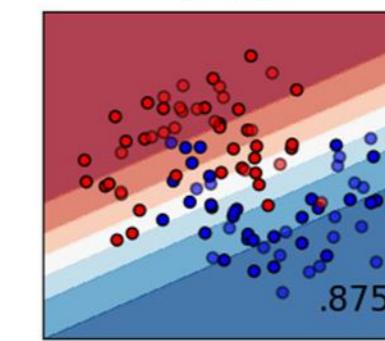
$C = 0.32$



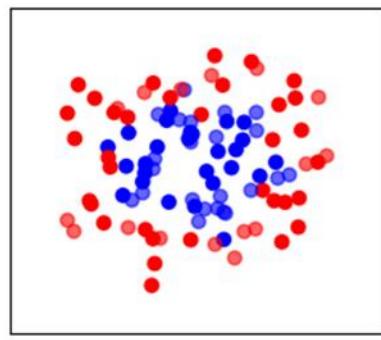
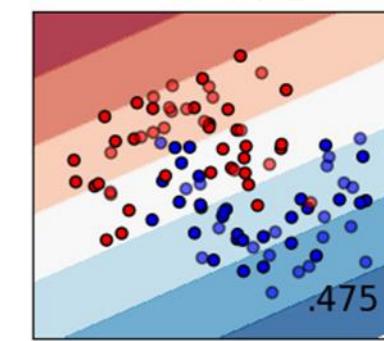
$C = 1.00$



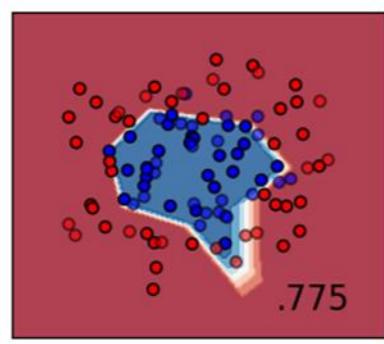
$C = 3.16$



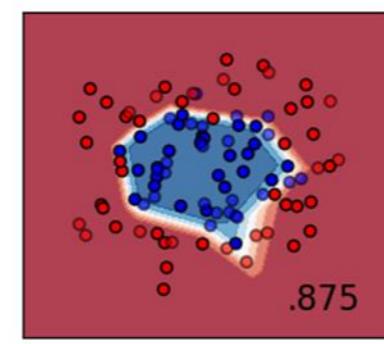
$C = 10.00$



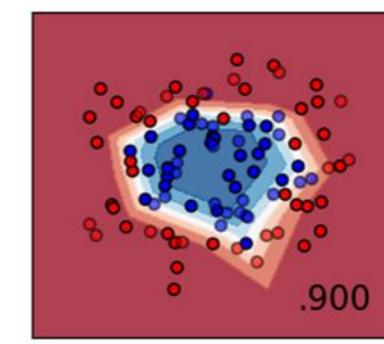
$C = 0.10$



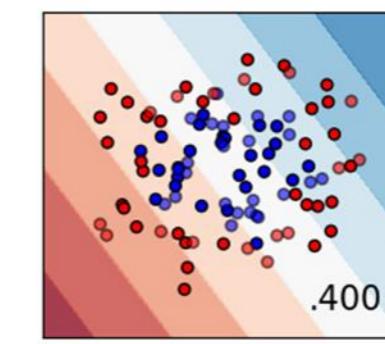
$C = 0.32$



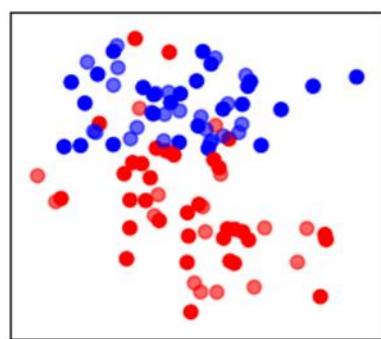
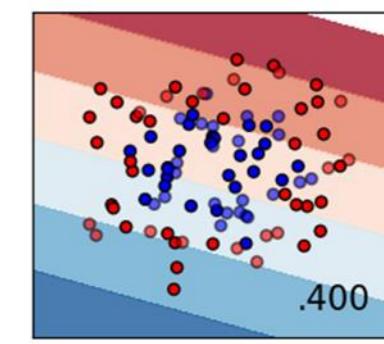
$C = 1.00$



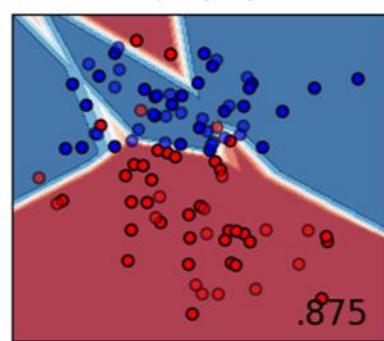
$C = 3.16$



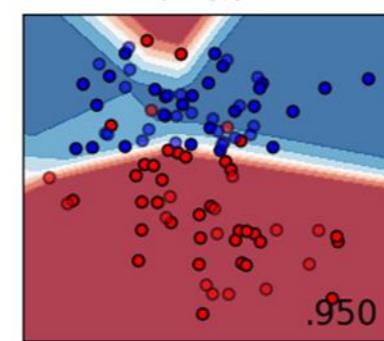
$C = 10.00$



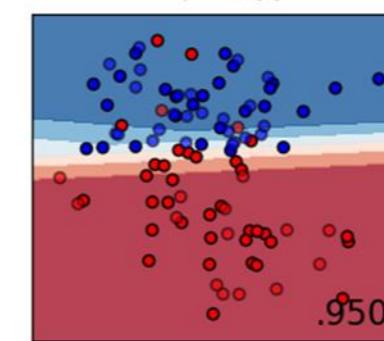
$C = 0.10$



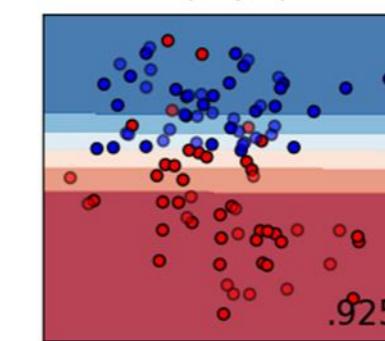
$C = 0.32$



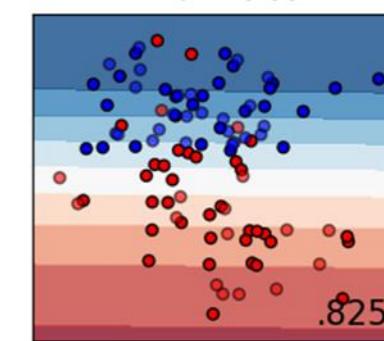
$C = 1.00$



$C = 3.16$



$C = 10.00$



## ¿Cómo queda entonces el problema de aprendizaje de un MLP?

- Al igual que un Perceptron, podemos definir fácilmente una función de pérdida:

$$J(W) = \frac{1}{2n} \sum_{i=1}^n (f(x_i; W) - y_i)^2$$

- Sin embargo, acá debemos considerar 3 nuevos elementos:

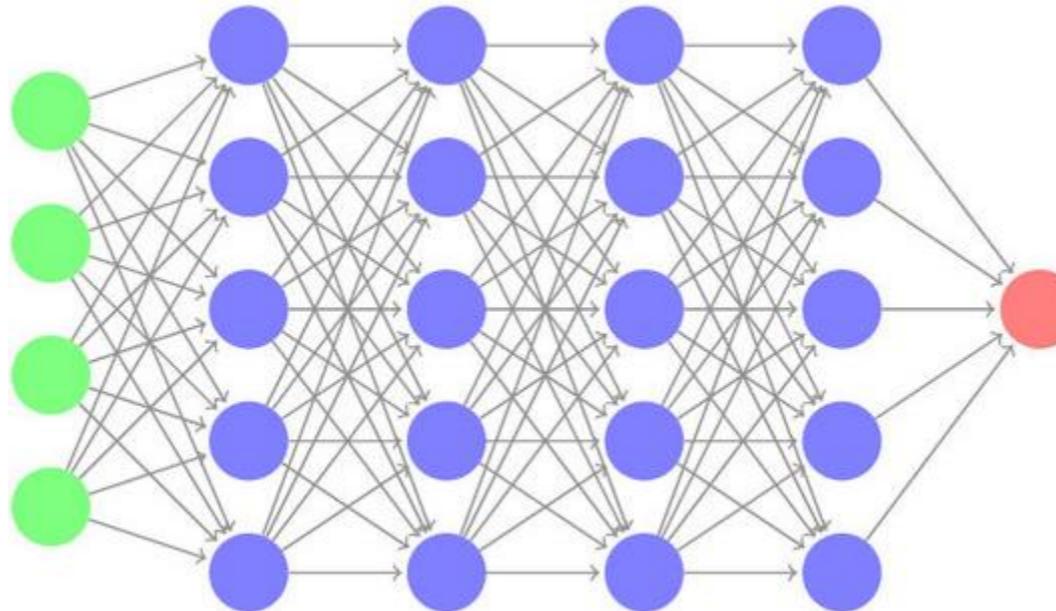
1. Ahora tenemos muchos más parámetros que antes, ¿cómo puedo controlar la complejidad?

Podemos tomar prestados conceptos de los SVM:

$$J(W) = \frac{C}{2} \|W\|^2 + \frac{1}{2n} \sum_{i=1}^n (f(x_i; W) - y_i)^2$$

2. ¿Cómo minimizo la función objetivo?

Cálculo de la derivada no es tan directo como con un Perceptron



Al tener composición de funciones, debemos aplicar de manera cuidadosa la regla de la cadena



1958 Perceptron

1974 Backpropagation



Handwritten Recognition



General Object  
Detection  
2008

No more funds

awkward silence (AI Winter)

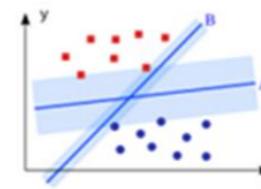
1969

Perceptron criticized



1995

SVM reigns



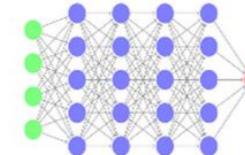
2005

Human  
Detection



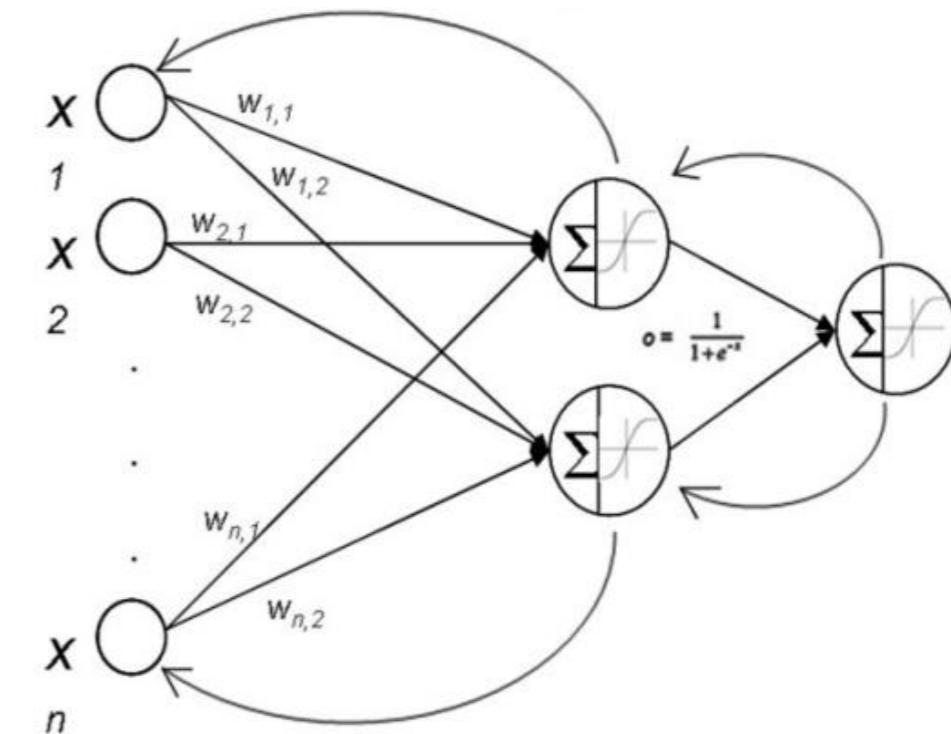
2012

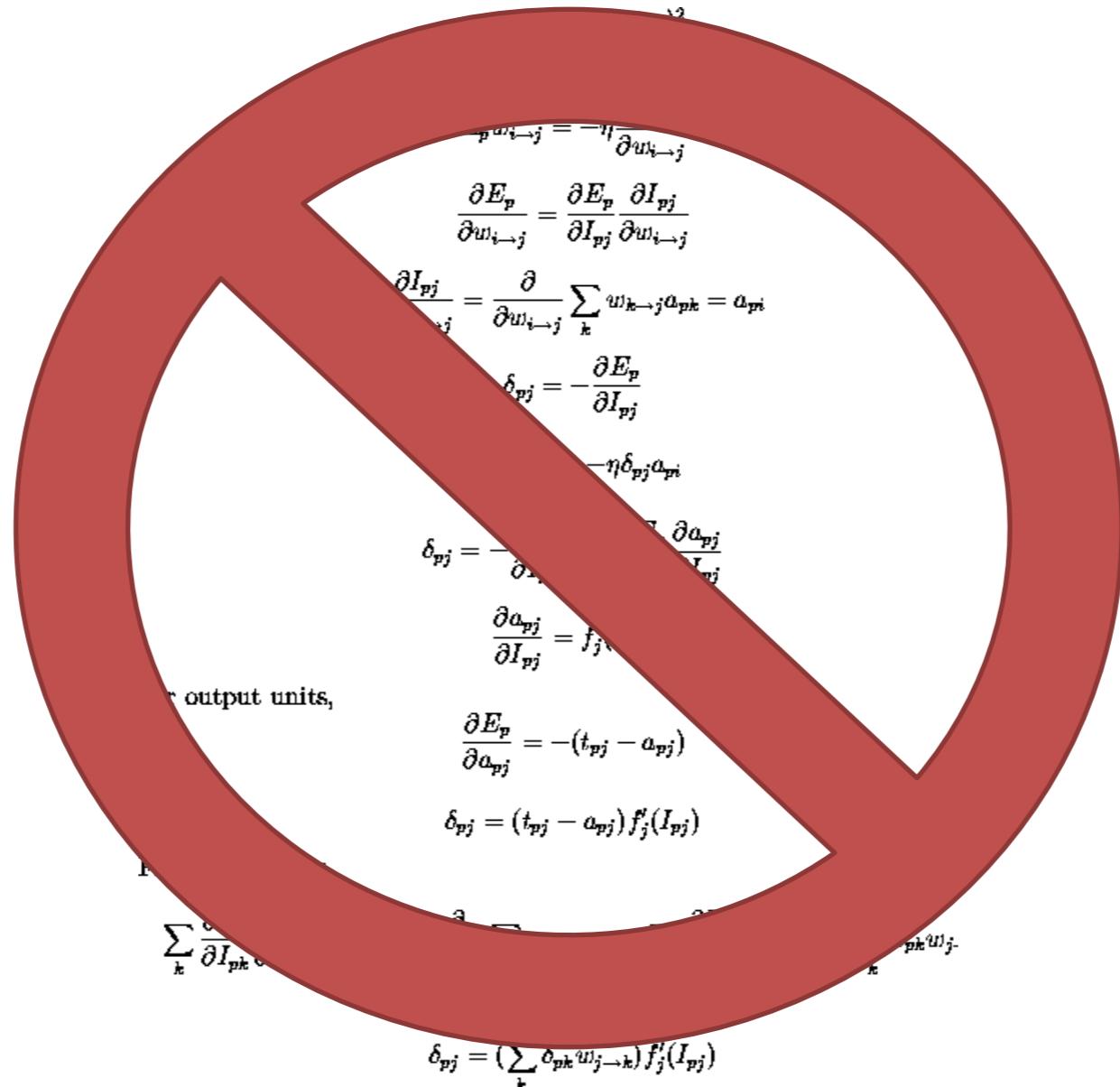
Deep Learning  
Rises



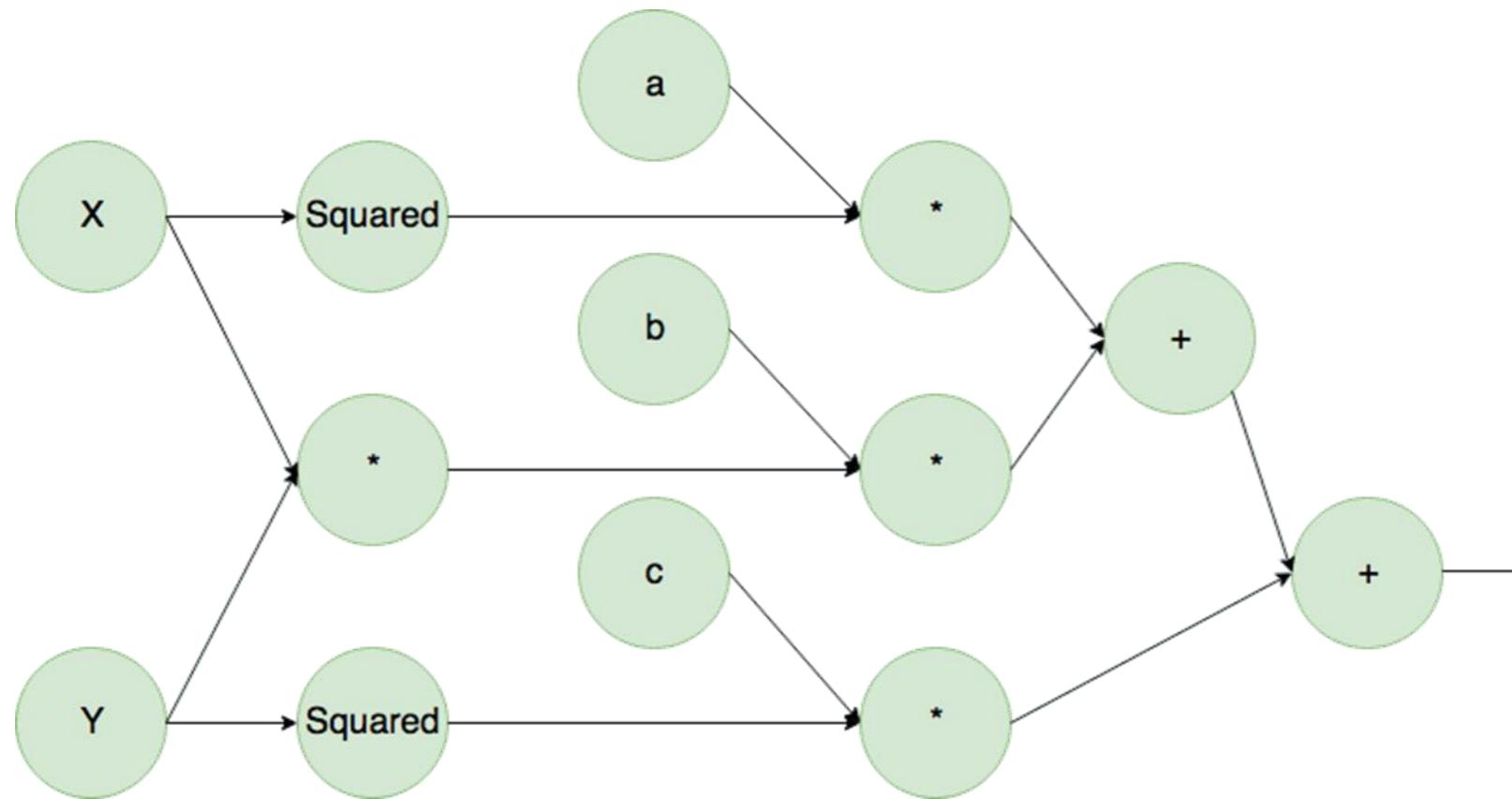
Backpropagation permite entrenar MLPs al propagar la señal del error a todos los nodos

- No es otra cosa que la aplicación recursiva de la **regla de la cadena**.
- En conjunto con el método del descenso del gradiente, permite teóricamente entrenar **redes de profundidad arbitraria**.
- Aplicable a cualquier tipo de red y función diferenciable.



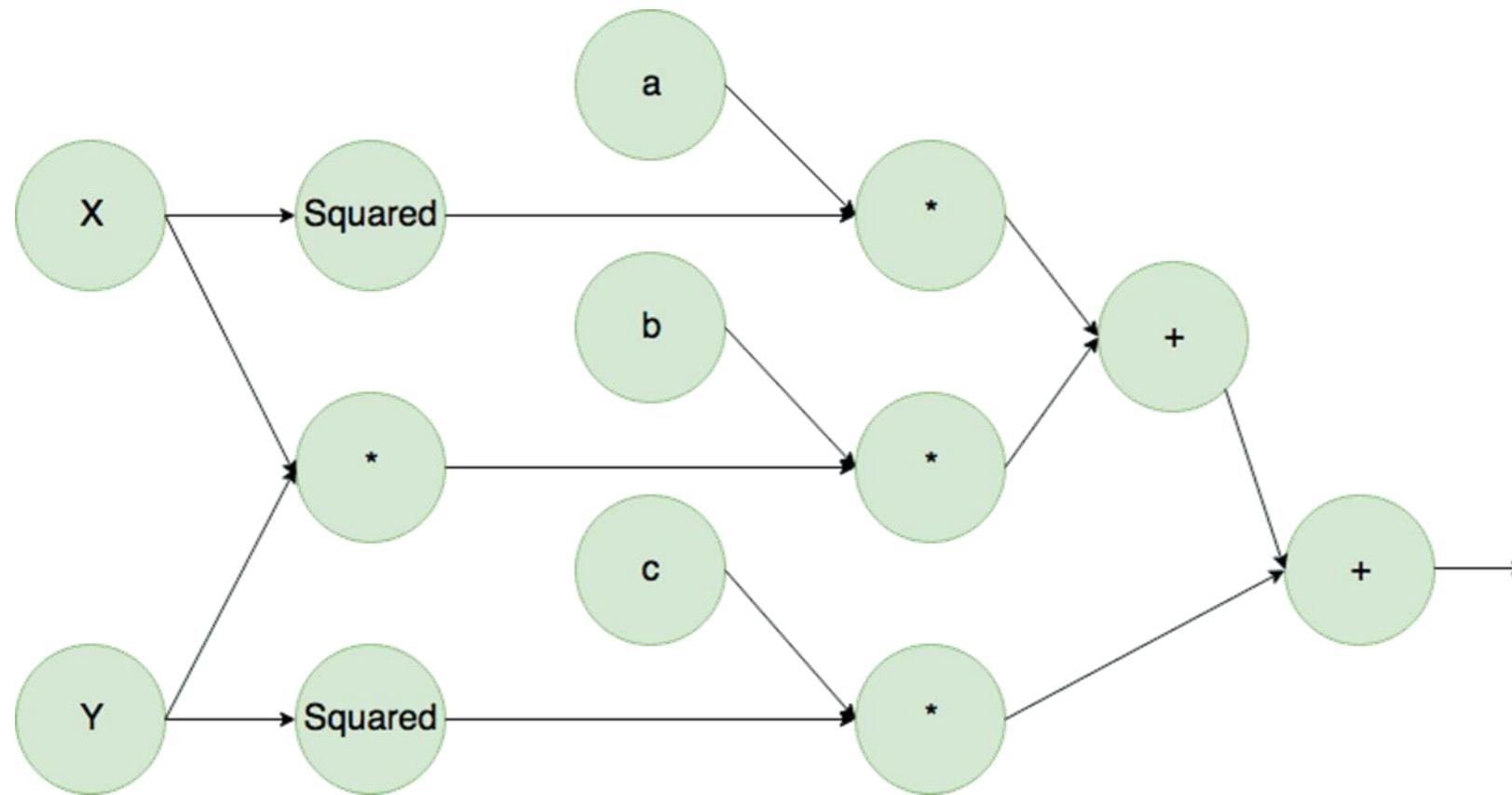


Ahora cambiaremos un poco de perspectiva sobre las redes, y comenzaremos a hablar de **programación diferenciable** y **grafos de cómputo**

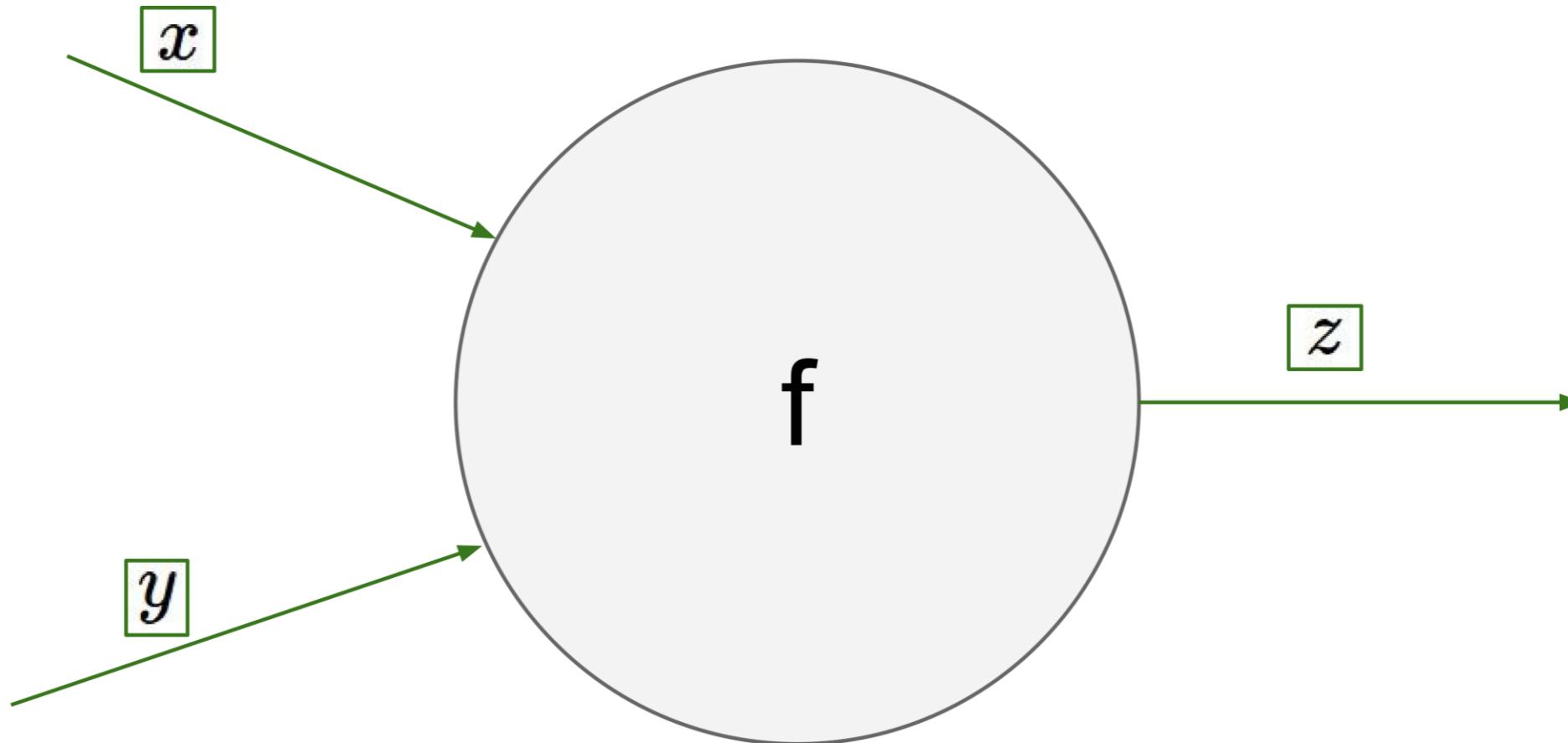


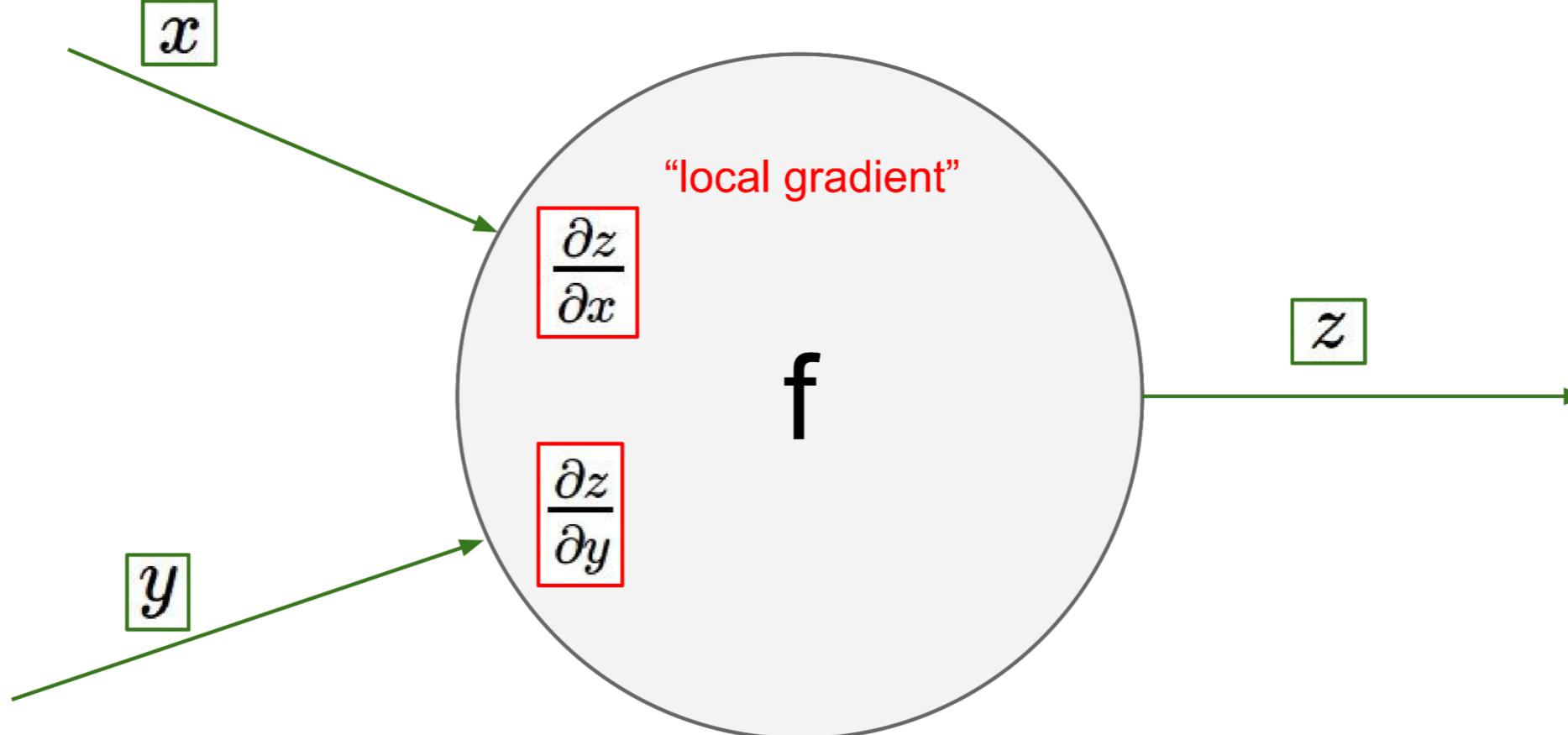
¿Qué función representa este grafo?

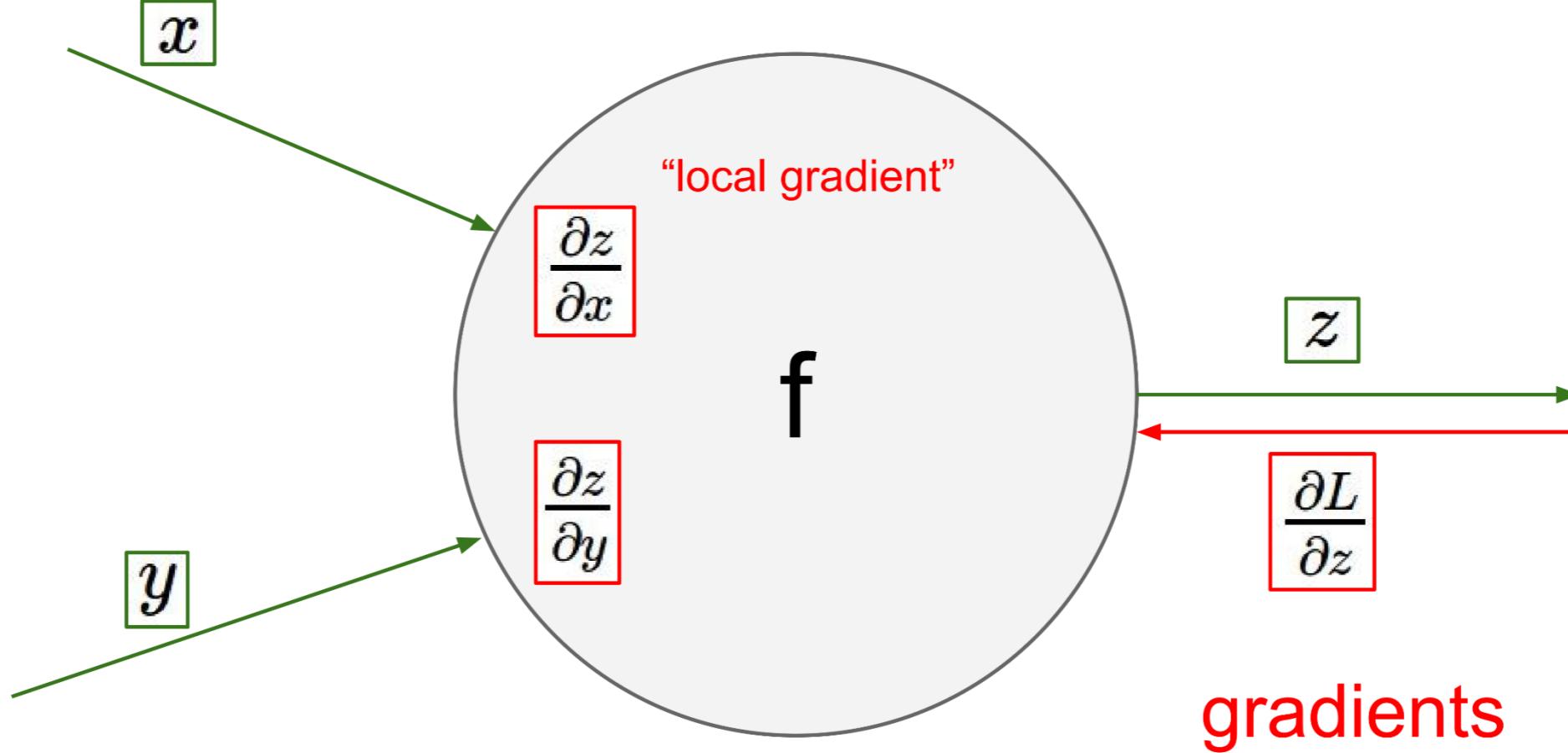
Un grafo de cómputo representa **todas las operaciones** realizadas en una función para obtener su salida

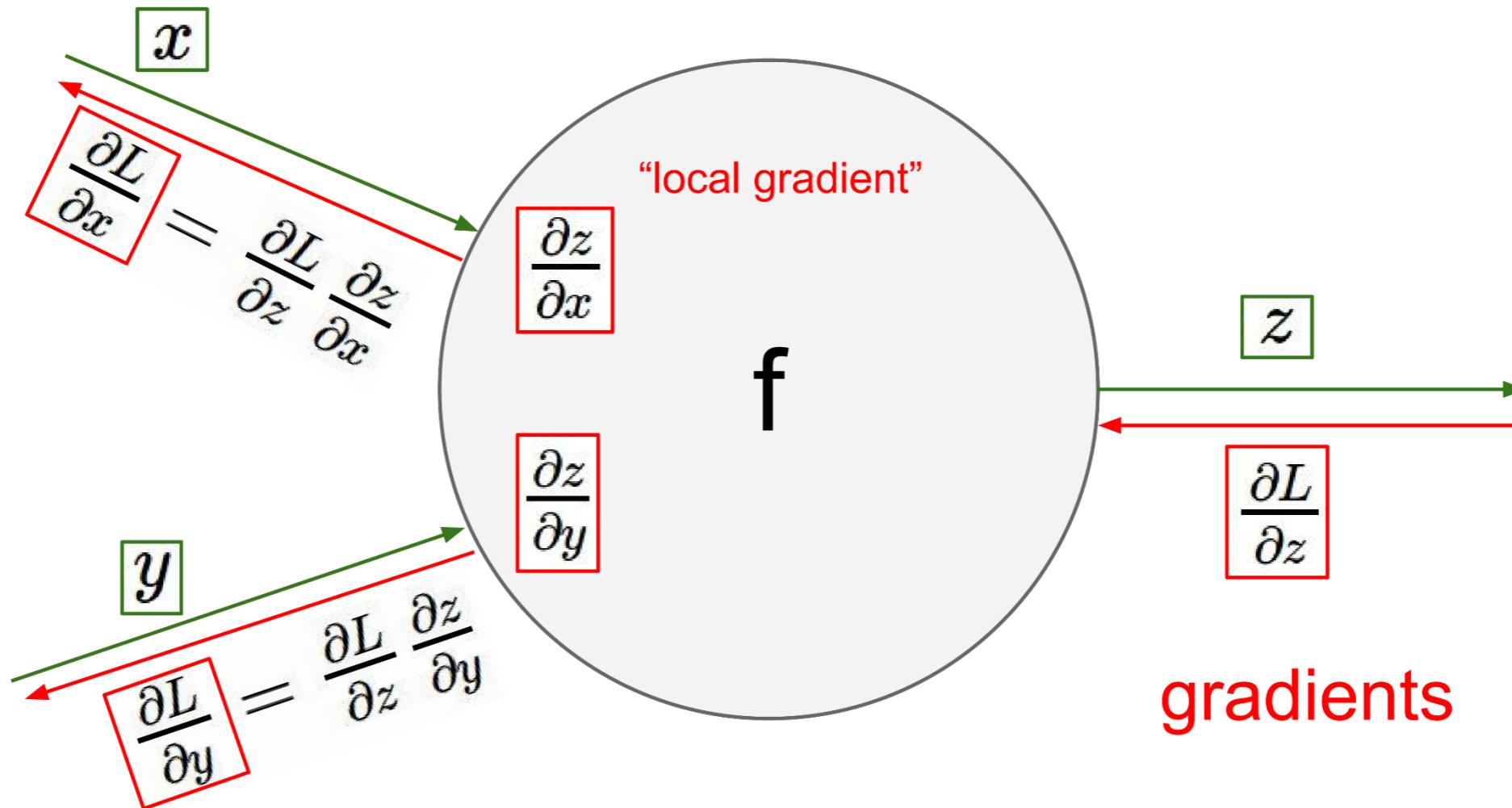


¿Y qué relación tiene esto con **Deep Learning** y **backprop**?



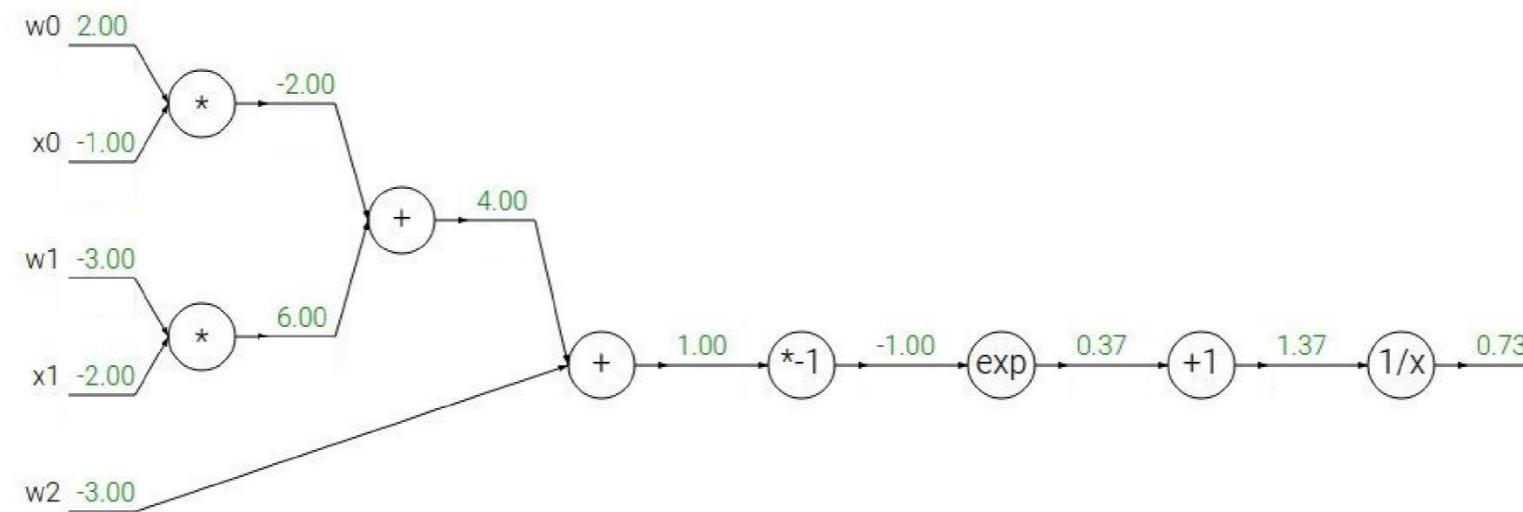






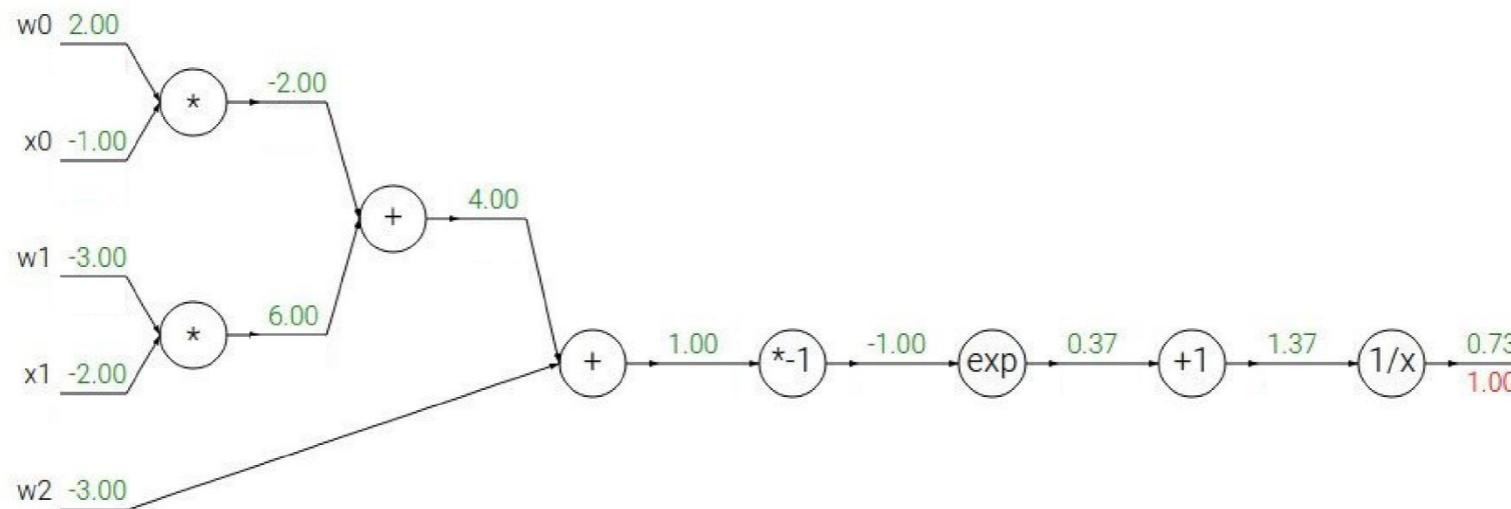
Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

$\rightarrow$

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

$\rightarrow$

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$\rightarrow$

$$\frac{df}{dx} = -1/x^2$$

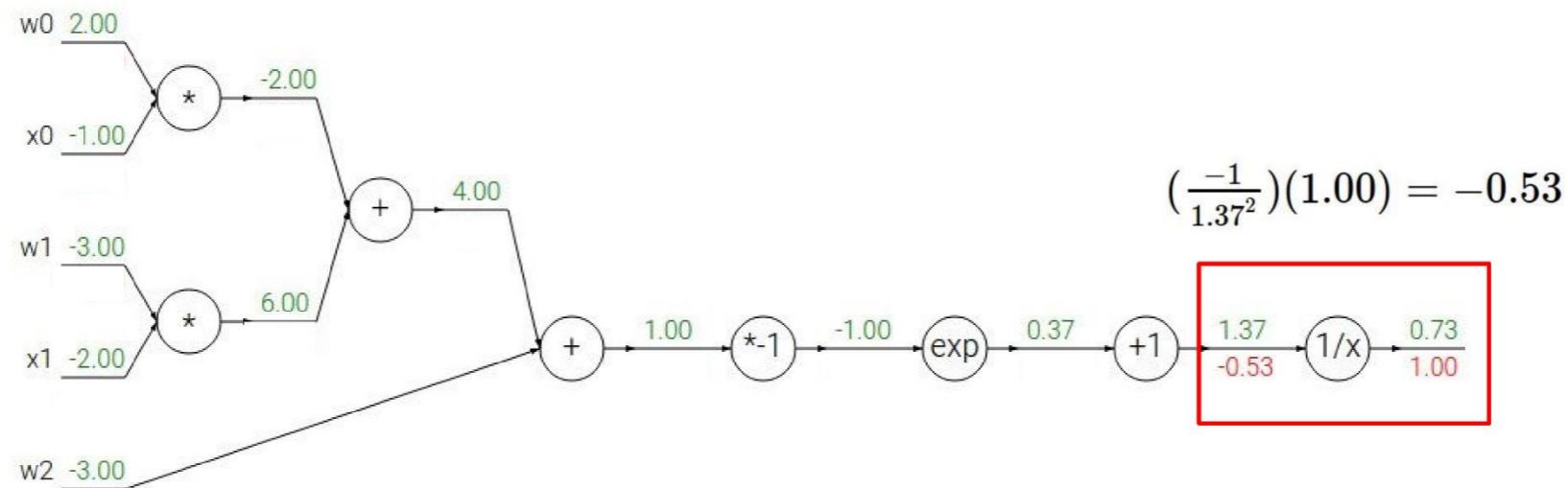
$$f_c(x) = c + x$$

$\rightarrow$

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

$\rightarrow$

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

$\rightarrow$

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$\rightarrow$

$$\frac{df}{dx} = -1/x^2$$

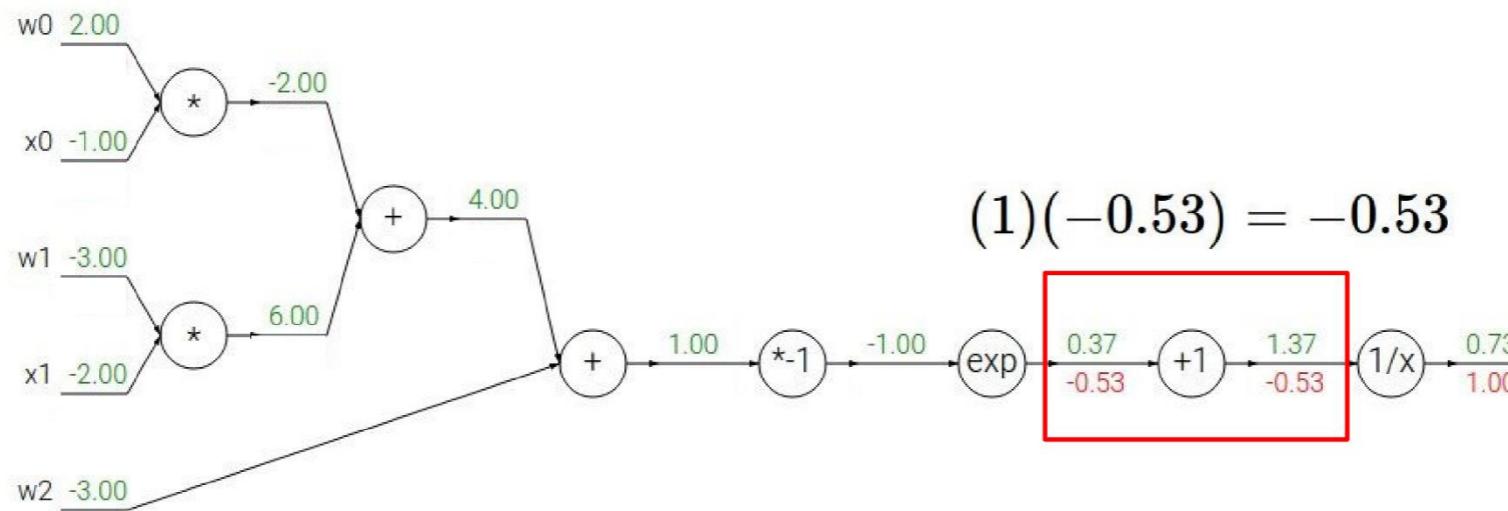
$$f_c(x) = c + x$$

$\rightarrow$

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

$\rightarrow$

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

$\rightarrow$

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$\rightarrow$

$$\frac{df}{dx} = -1/x^2$$

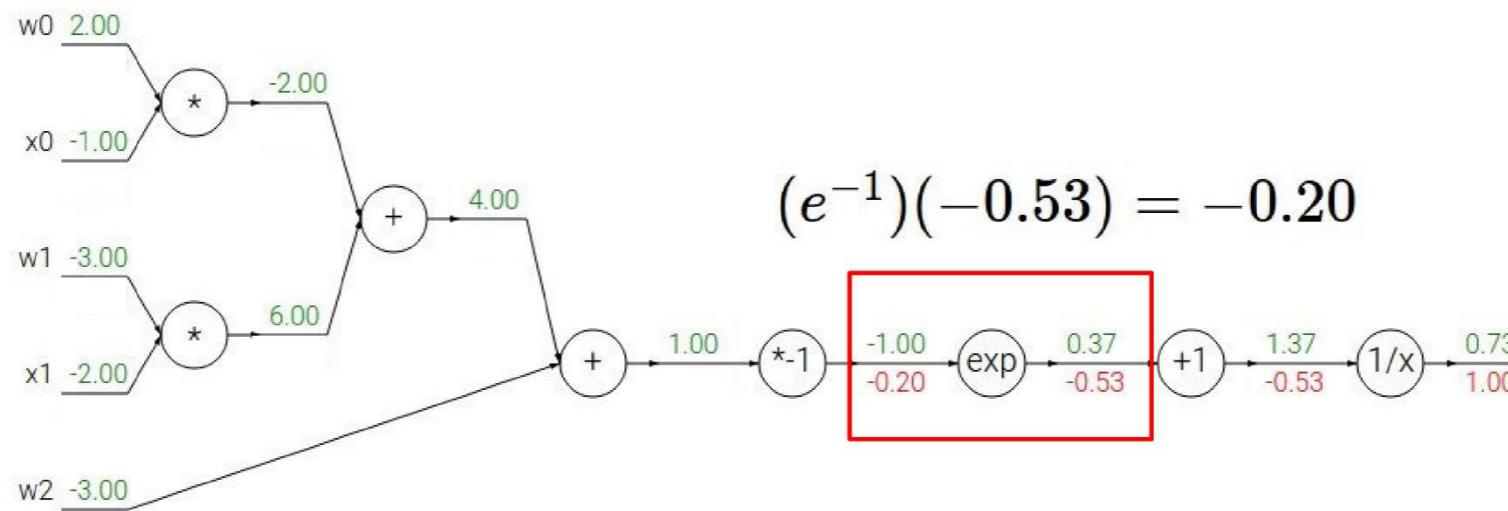
$$f_c(x) = c + x$$

$\rightarrow$

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

$$\rightarrow$$

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$$\rightarrow$$

$$\frac{df}{dx} = -1/x^2$$

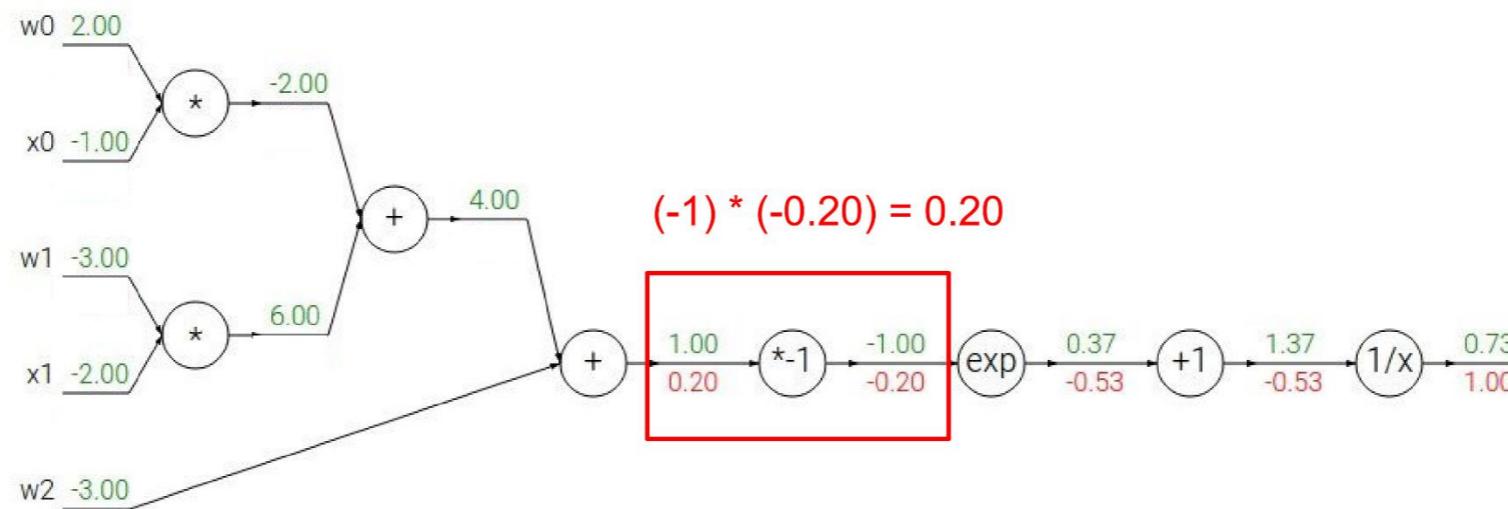
$$f_c(x) = c + x$$

$$\rightarrow$$

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

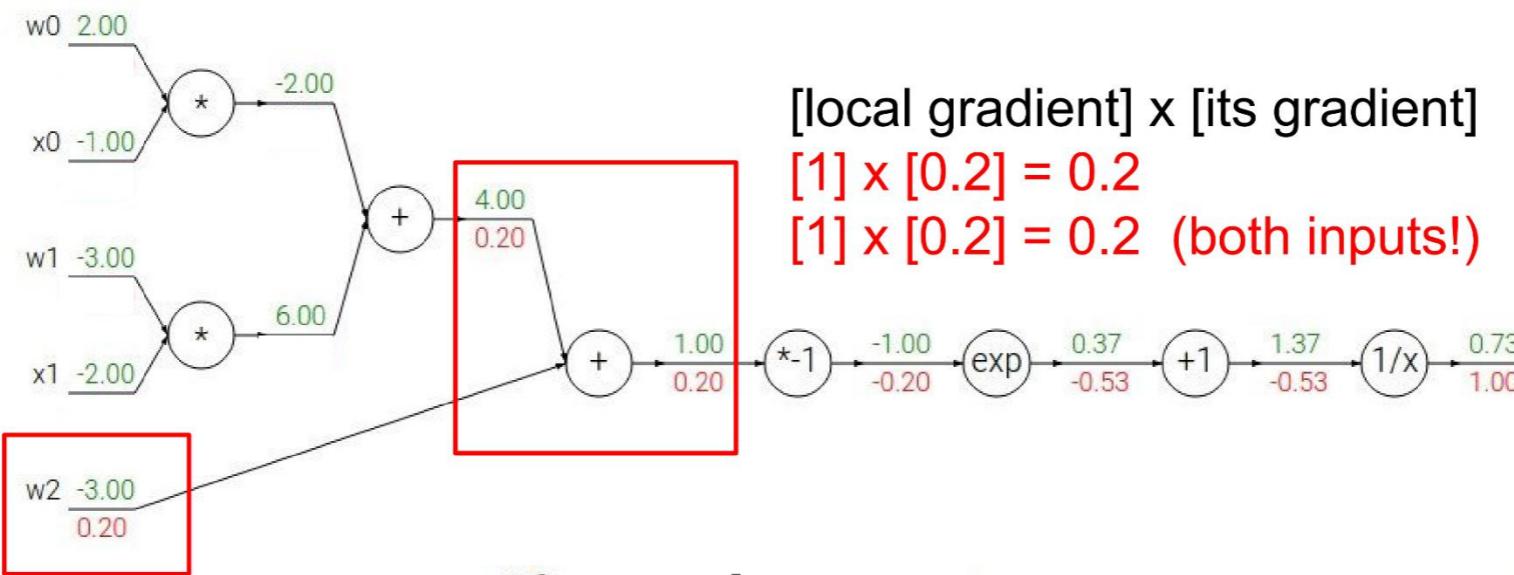
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

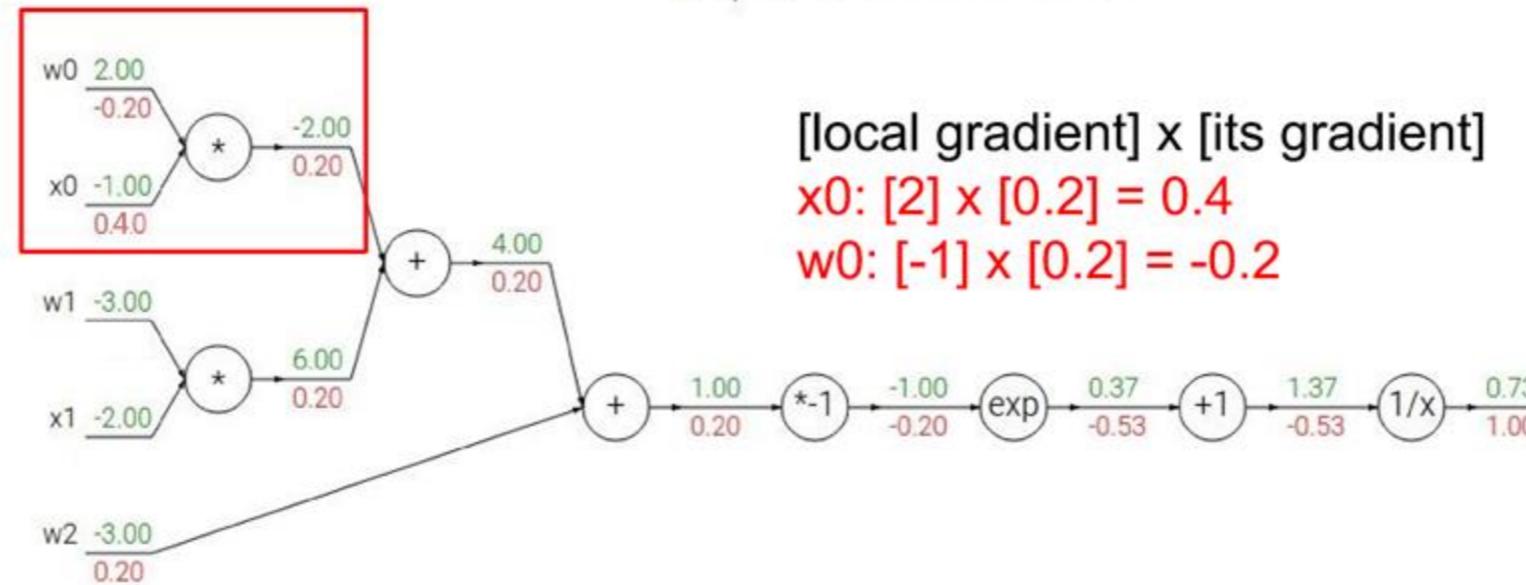
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



[local gradient] x [its gradient]

$$x_0: [2] \times [0.2] = 0.4$$

$$w_0: [-1] \times [0.2] = -0.2$$

$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

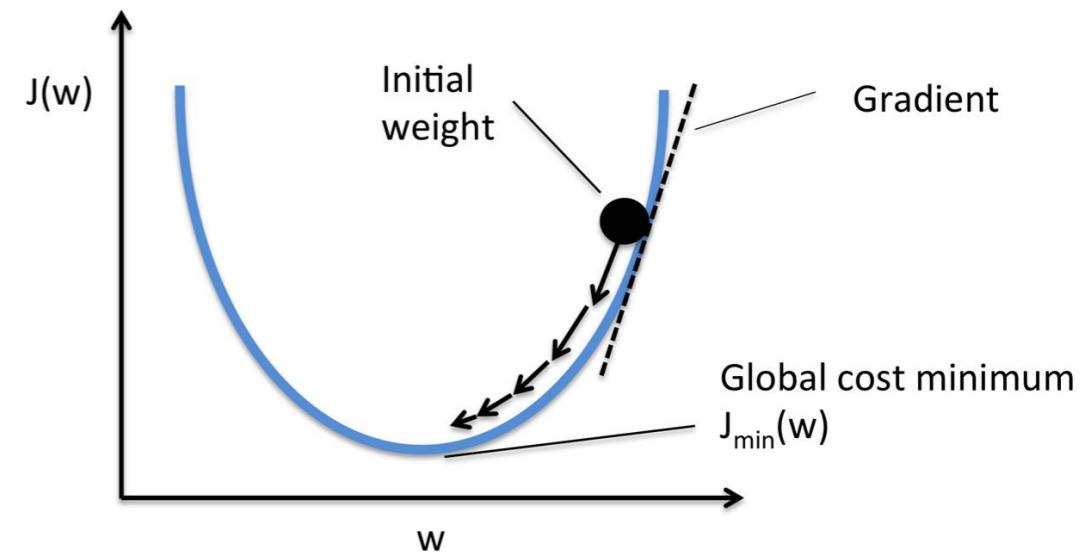
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

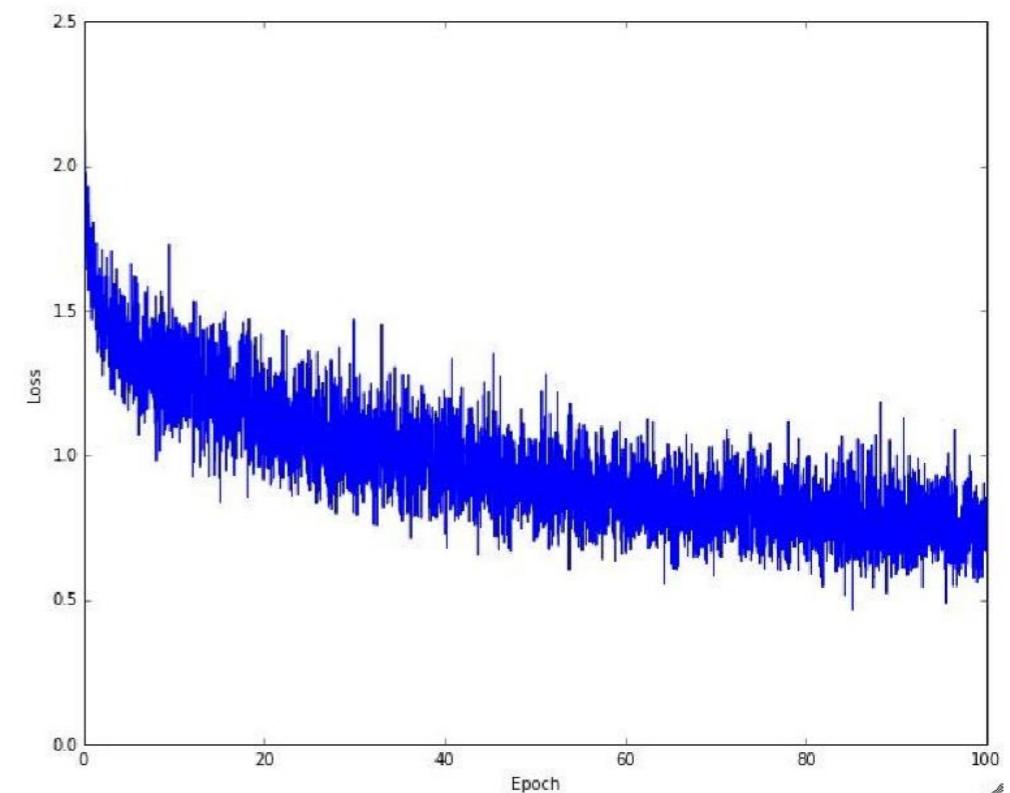
## Descenso de gradiente puede ser demasiado demandante computacionalmente

- Para redes y/o conjuntos de datos grandes, el cálculo de la función y su derivada toman un tiempo considerable.
- Si bien esto puede acelerarse utilizando GPUs, este tipo de escalamiento es costoso.
- ¿Existe algún supuesto que podamos utilizar sobre los datos que pudiera ayudarnos?

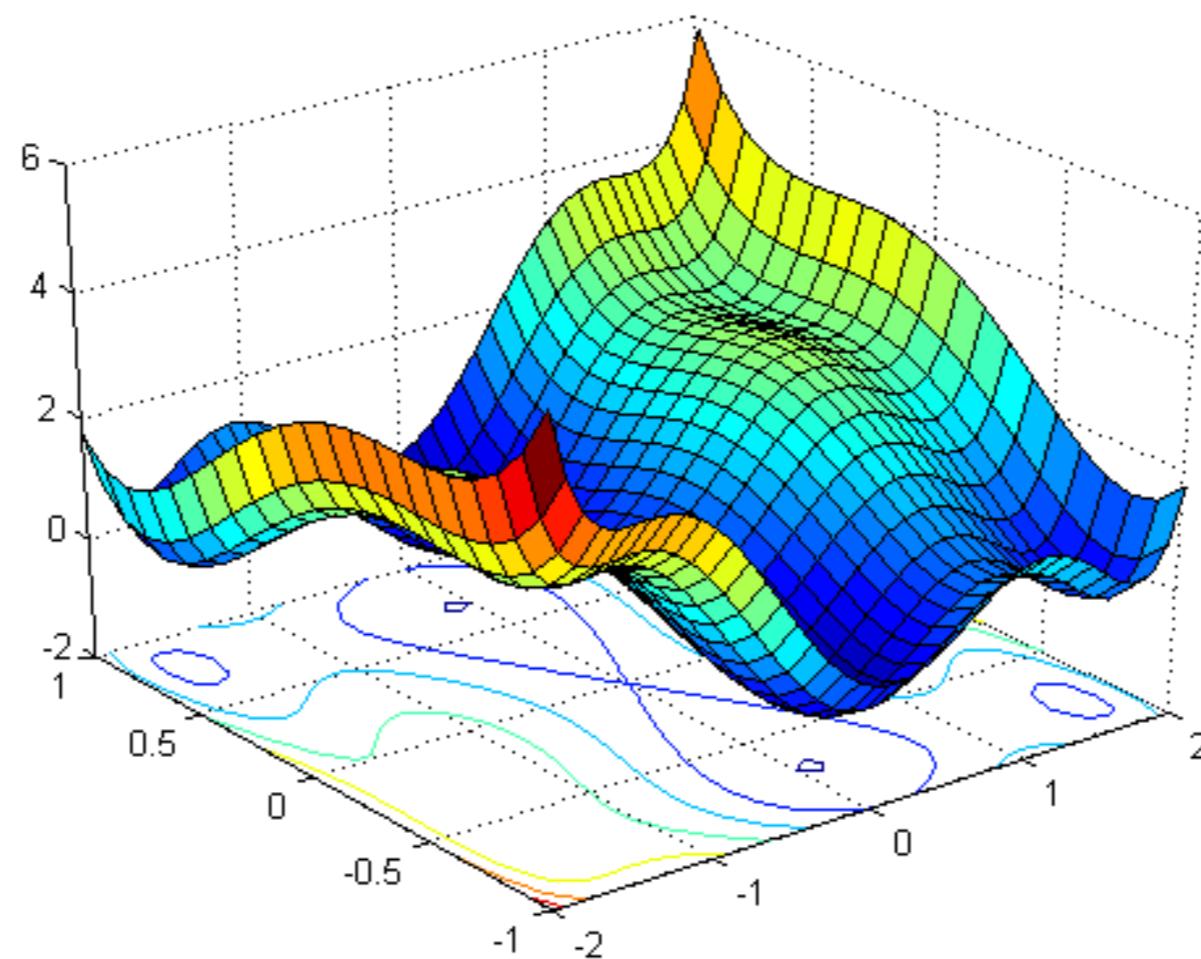


## Stochastic Gradient Descent (SGD) aprovecha la fuerza de los promedios

- Consiste en usar un mini-batch de ejemplos para calcular el gradiente para cada descenso, en vez de todos los ejemplos.
- Esta aproximación es un estimador insesgado de la esperanza del gradiente real.
- Existen garantías teóricas de convergencia (tomando un learning rate suficientemente pequeño)
- Tamaños comunes para un mini-batch son 32/64/128/256
- Estocasticidad permite incluso evitar algunos puntos críticos de la función, lo que nos lleva el siguiente punto...

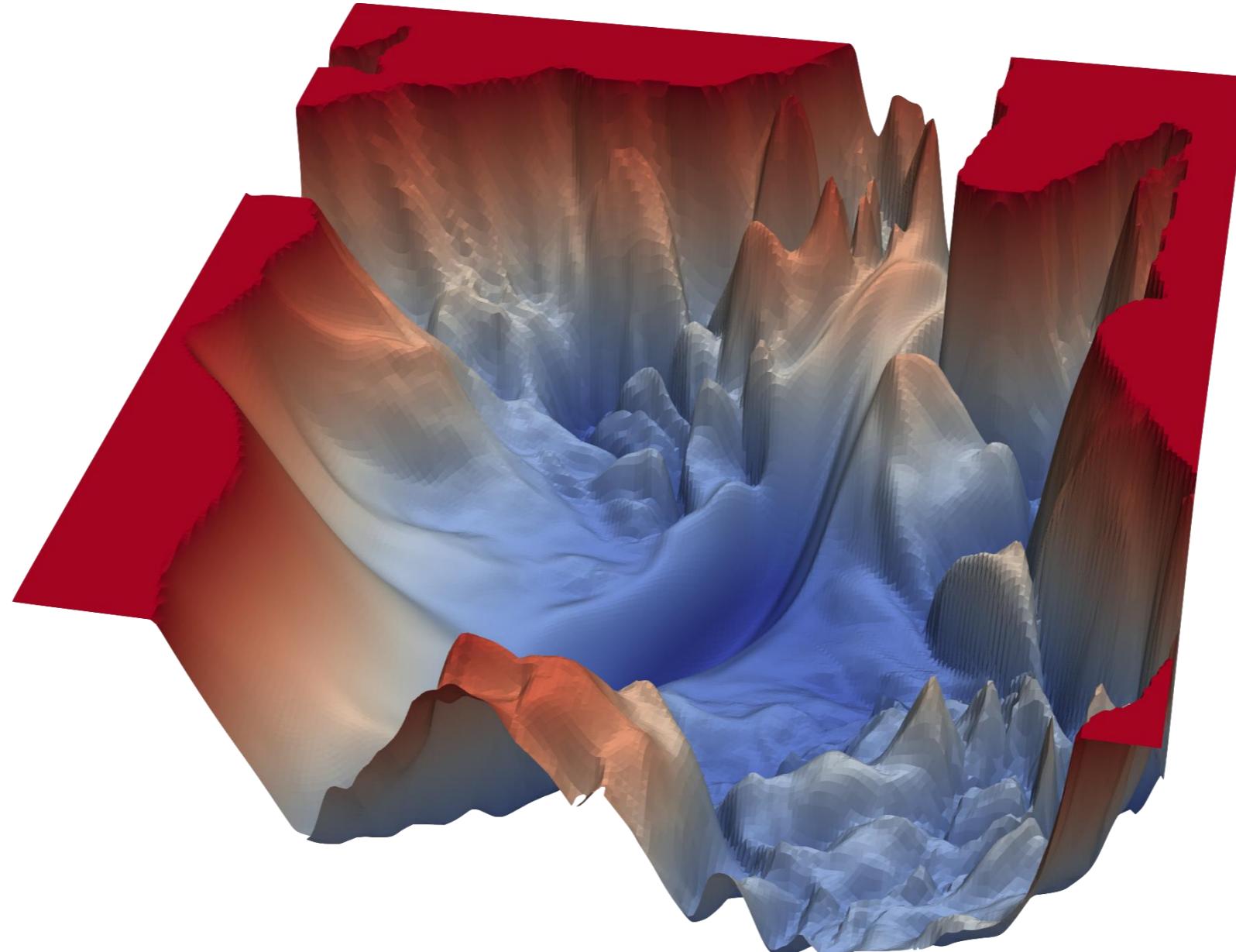


Debido a la estructura de las redes, función objetivo es no convexa



Ni siquiera es no convexa pero suave

La realidad es mucho peor (en otras palabras, el descenso de gradiente estocástico no sirve)





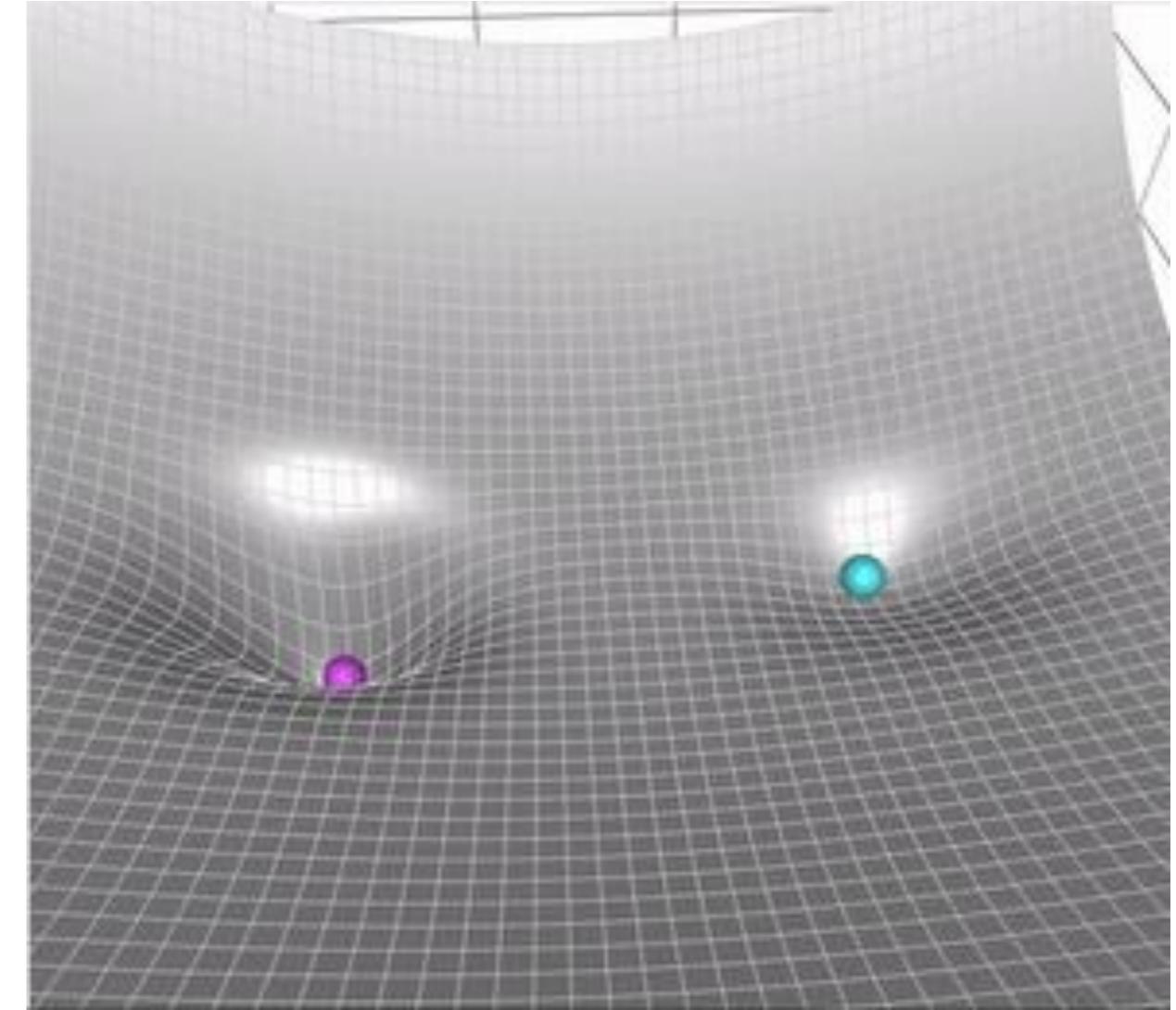
<https://youtu.be/83827jaSsGU>

<https://losslandscape.com/>

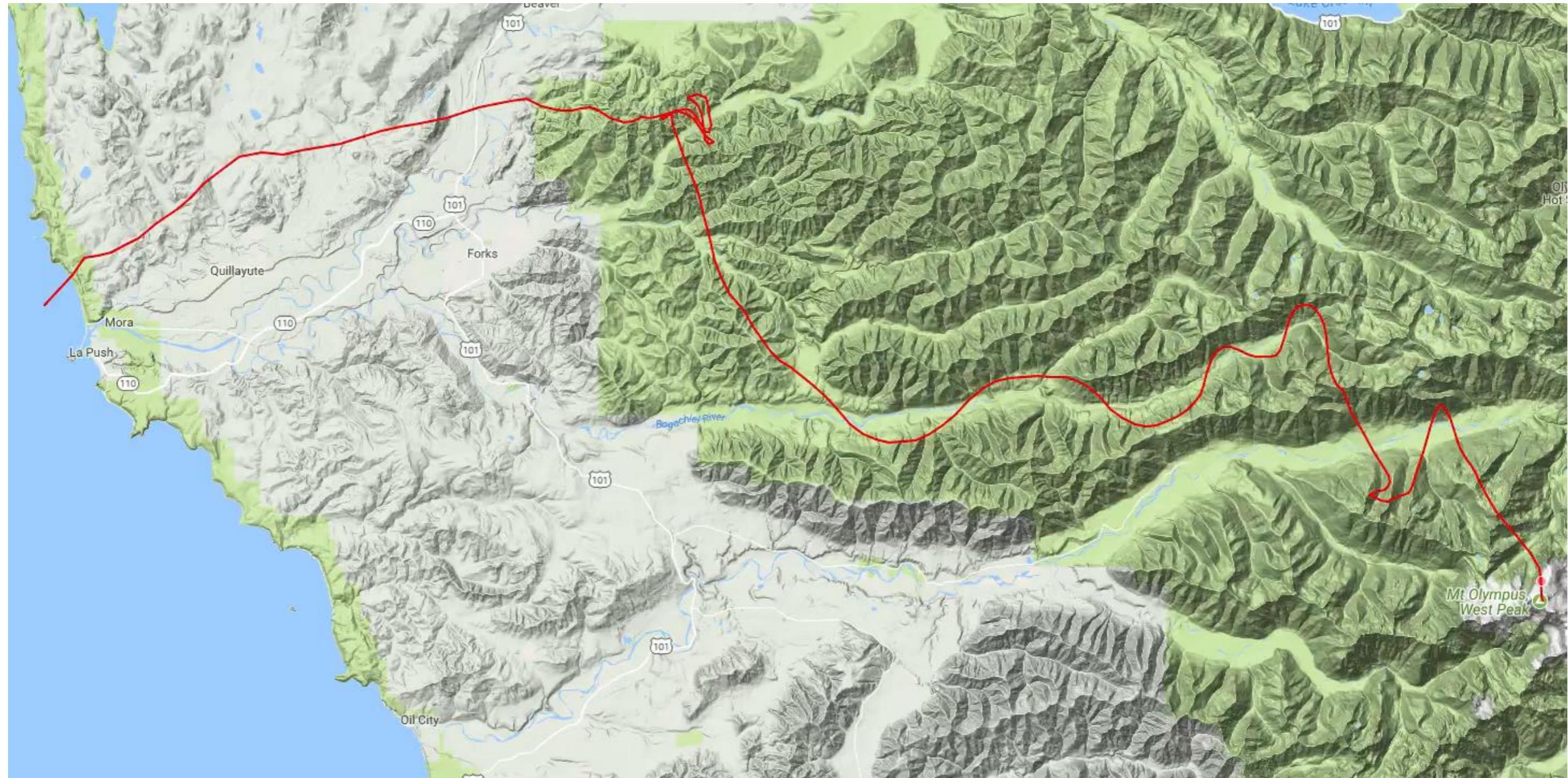
Momentum permite aminorar riesgo de quedar pegado en puntos críticos

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta_t)$$

$$\theta_{t+1} = \theta_t - v_t$$



Un ejemplo muy “bonito” puede construirse utilizando datos geográficos de elevación



<sup>1</sup><https://fosterelli.co/executing-gradient-descent-on-the-earth>

## ¿Cómo queda entonces el problema de aprendizaje de un MLP?

- Al igual que un Perceptron, podemos definir fácilmente una función de pérdida:

$$J(W) = \frac{1}{2n} \sum_{i=1}^n (f(x_i; W) - y_i)^2$$

- Sin embargo, acá debemos considerar 3 nuevos elementos:

1. Ahora tenemos muchos más parámetros que antes, ¿cómo puedo controlar la complejidad?

Podemos tomar prestados conceptos de los SVM:

$$J(W) = \frac{C}{2} \|W\|^2 + \frac{1}{2n} \sum_{i=1}^n (f(x_i; W) - y_i)^2$$

2. ¿Cómo minimizo la función objetivo?

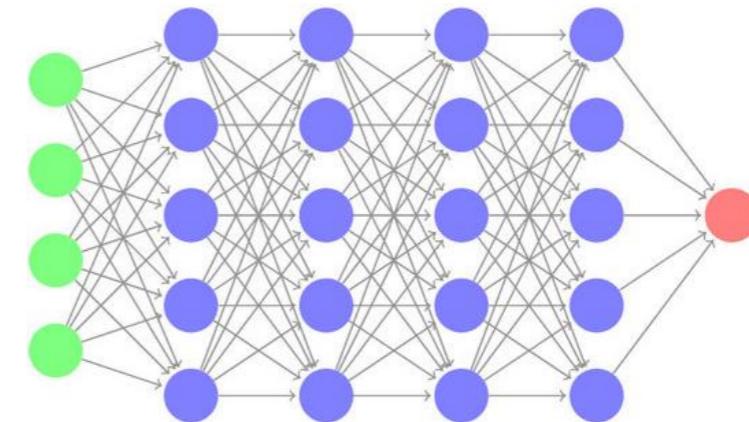
Backpropagation sobre un grafo de cómputo + SGD + momentum

3. ¿Qué límites tiene un MLP en cuanto a lo que puede aprender?

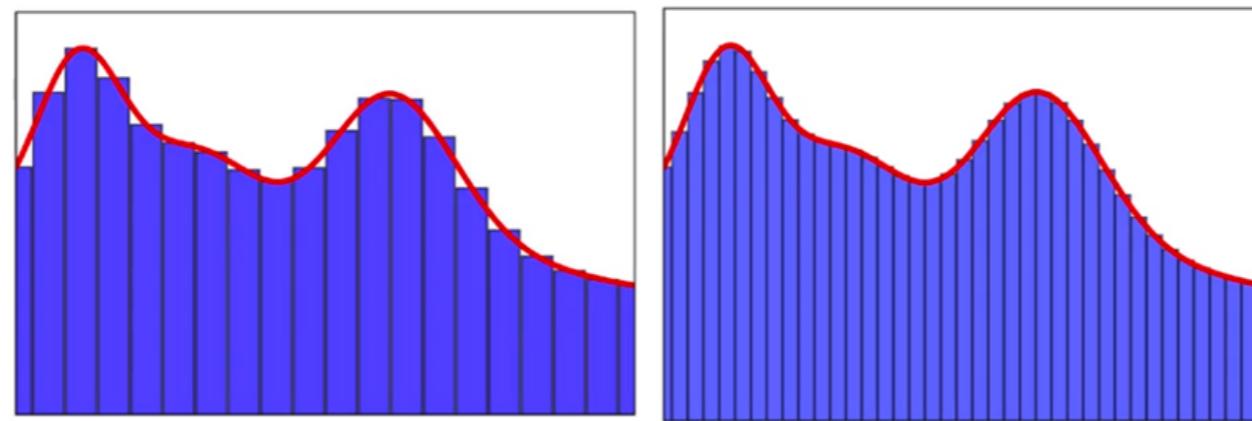
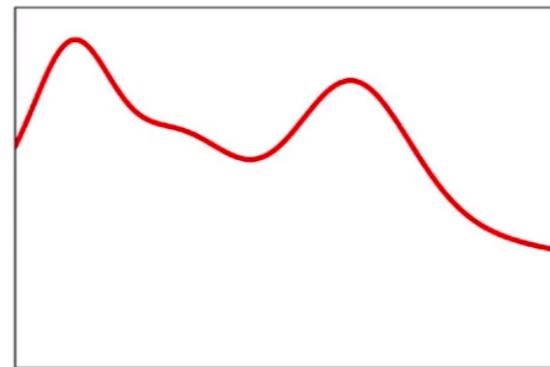
Spoiler: no tiene límites

## ¿Hasta dónde podemos llegar con un MLP?

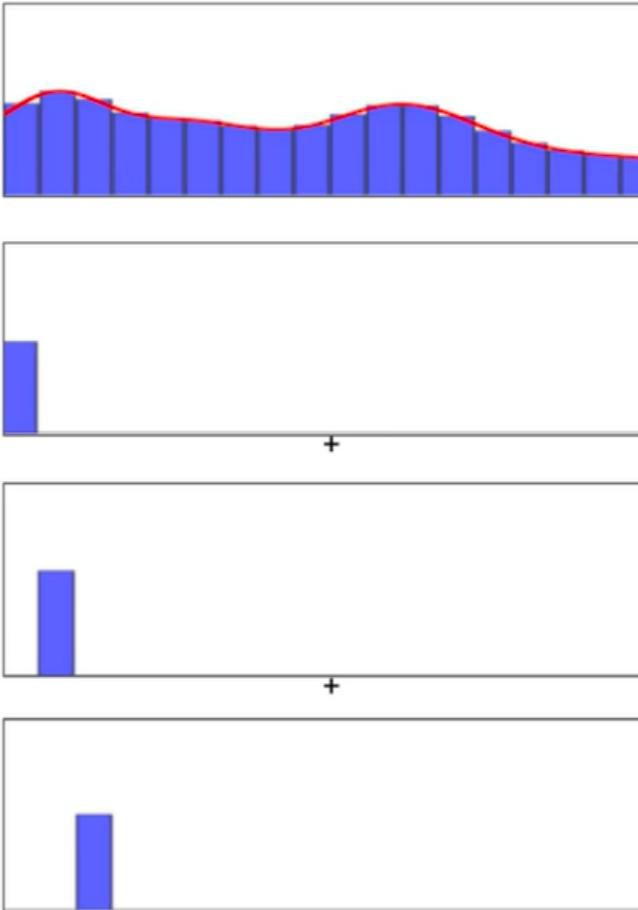
- De hecho, hasta donde uno quiera.
- De acuerdo al “*teorema de aproximación universal*”, un MLP con 1 capa oculta puede aproximar cualquier función continua, con precisión arbitraria.
- Además, un MLP con 2 capas ocultas es capaz de aproximar cualquier función, con precisión arbitraria.
- Una buena “mostración” del teorema puede verse en <https://www.youtube.com/watch?v=Ijqkc7OLenI>



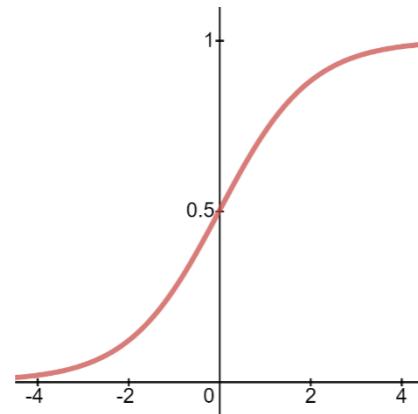
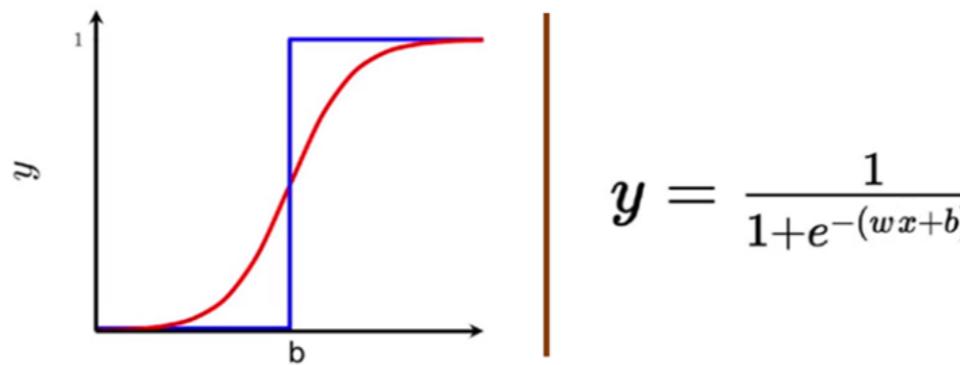
Veamos una pequeña “mostración” del teorema



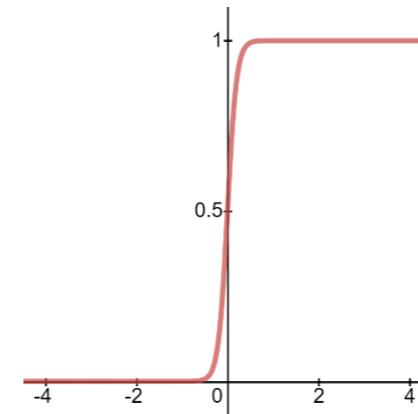
Veamos una pequeña “mostración” del teorema



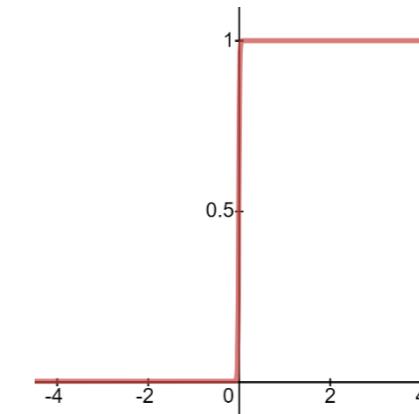
Veamos una pequeña “mostración” del teorema



$$w = 1$$

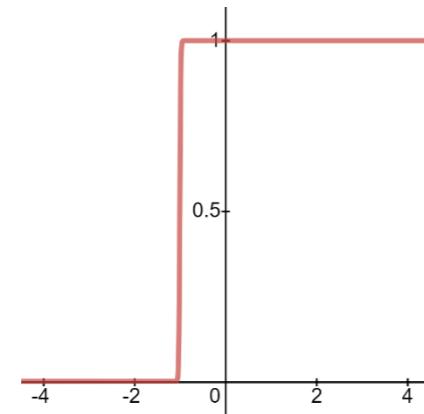
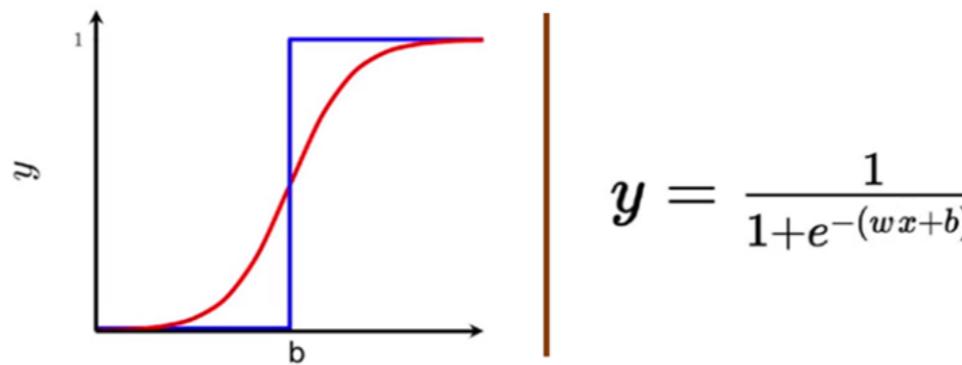


$$w = 10$$

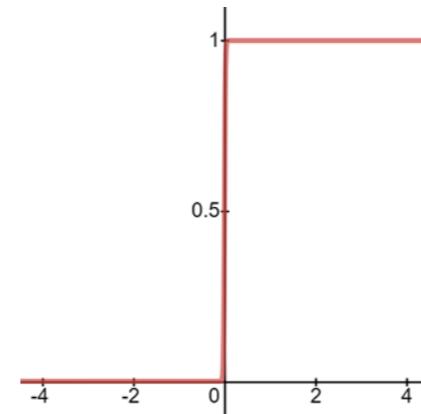


$$w = 100$$

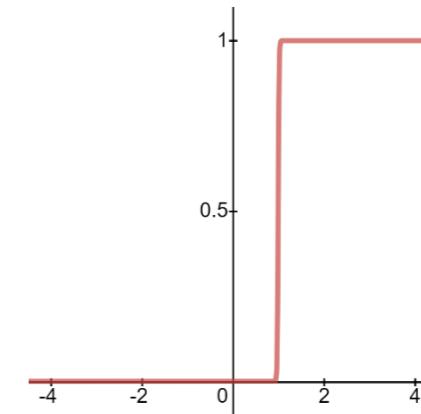
Veamos una pequeña “mostración” del teorema



$$w = 100, b = 100$$

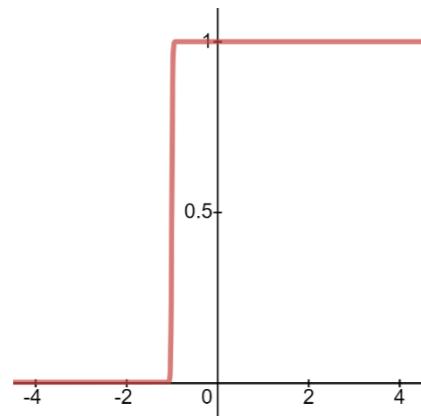


$$w = 100, b = 0$$

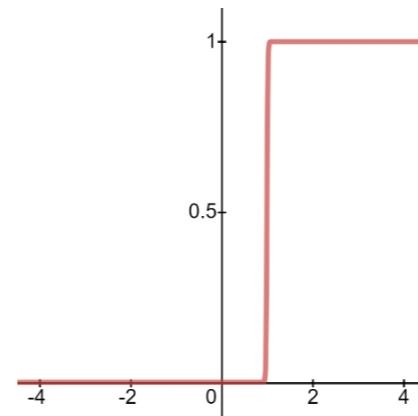


$$w = 100, b = -100$$

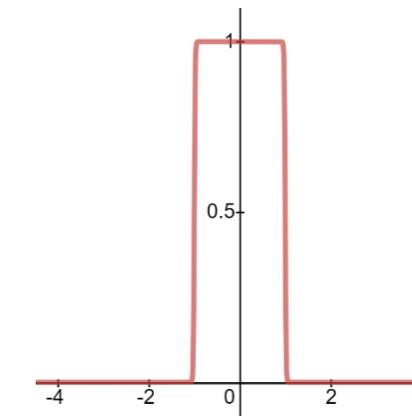
Veamos una pequeña “mostración” del teorema



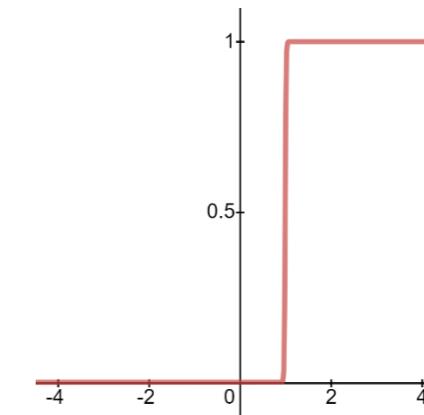
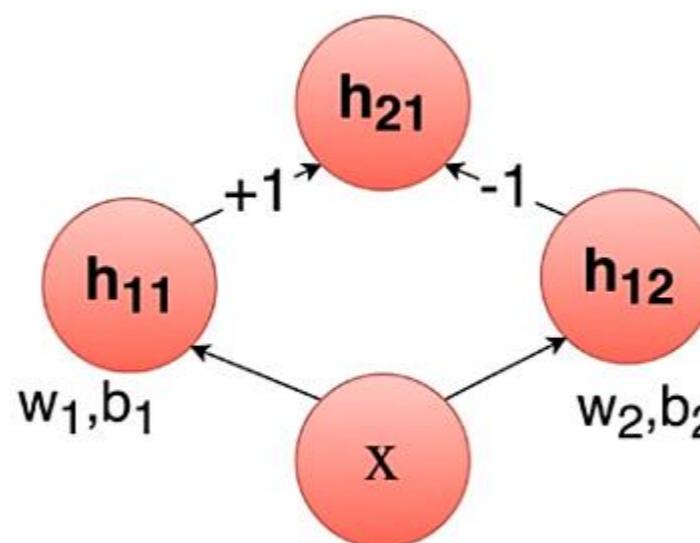
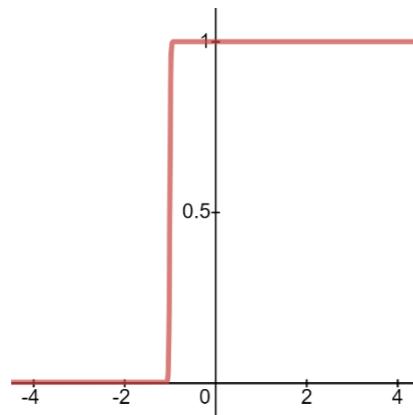
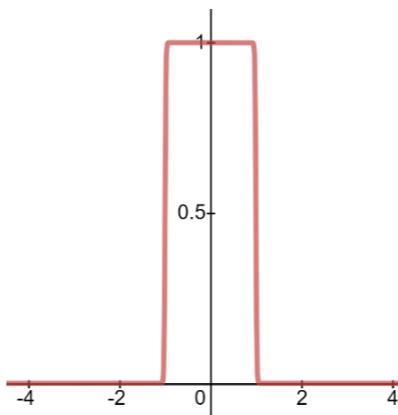
—



=



Veamos una pequeña “mostración” del teorema



## MLP son teóricamente todopoderosos

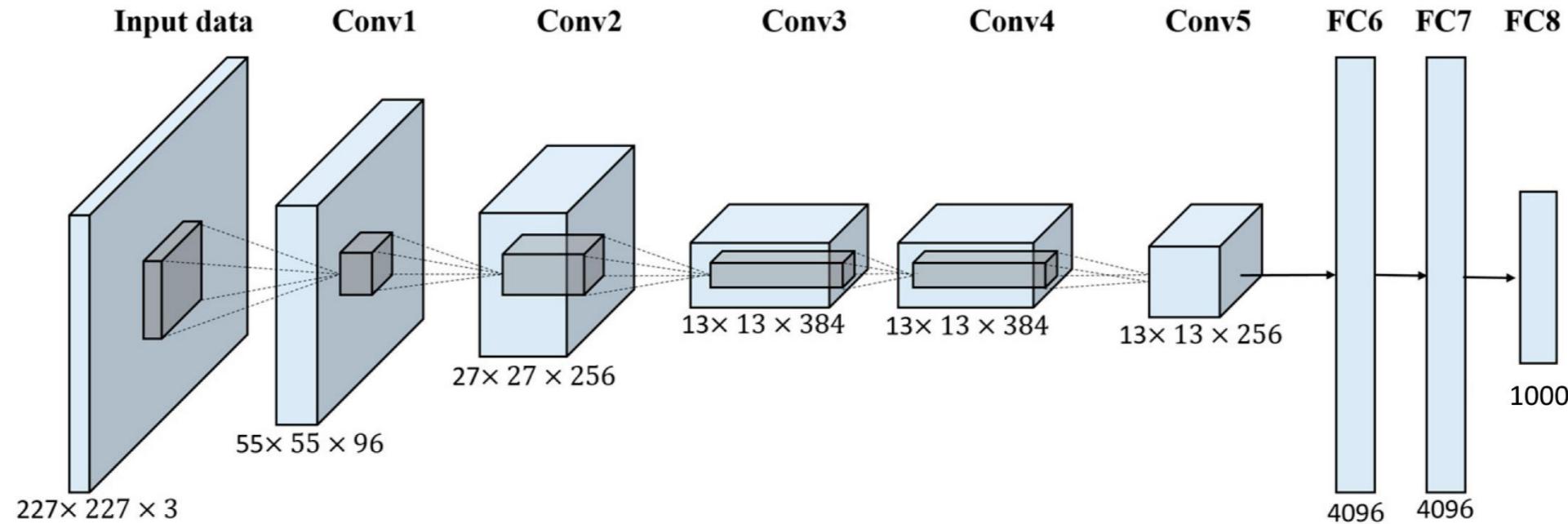
- En resumen, el teorema nos indica que, dada una función, siempre es posible construir un MLP que la estime con precisión arbitraria.
- Sin embargo, la cantidad de neuronas por capa va a depender del problema, o sea, puede ser potencialmente muy grande.
- Esta metodología no es práctica y es distinta de entrenar un MLP, pero al menos nos da una idea de una cota superior para el tamaño de un MLP.

Cerremos con un caso de estudio

## ImageNet Classification with Deep Convolutional Neural Networks

A. Krizhevsky, I. Sutskever, and G. E. Hinton (NeurIPS 2012)

AlexNet fue uno de los primeros ejemplos exitosos de ConvNets

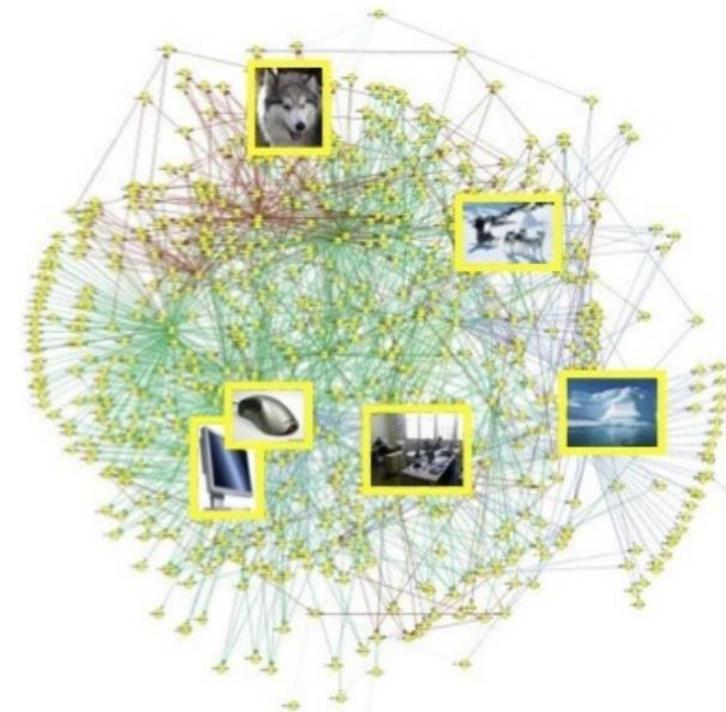


- Además de utilizar redes convoluciones, introdujo una serie de “trucos” muy útiles para el entrenamiento de redes.
- Sin embargo, el gran diferenciador fue el ser capaz de aprender efectivamente de un set de datos órdenes de magnitud mayor que los existentes hasta ese momento.

## Comencemos con los datos

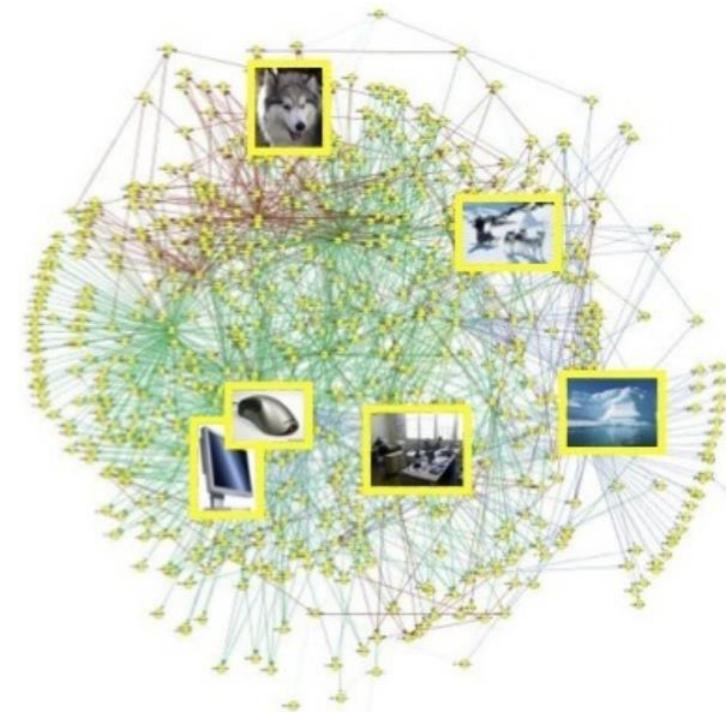
- La base para la gran mayoría de los trabajos iniciales en redes convolucionales para reconocimiento visual utiliza ImageNet.
- Este set de datos presenta más de 15M de imágenes divididas en 22K categorías.
- Además, imágenes están relacionadas por una estructura de grafo, que captura relaciones entre ellas.
- Las imágenes fueron recolectadas de la web y rotuladas utilizando Amazon Mechanical Turk.

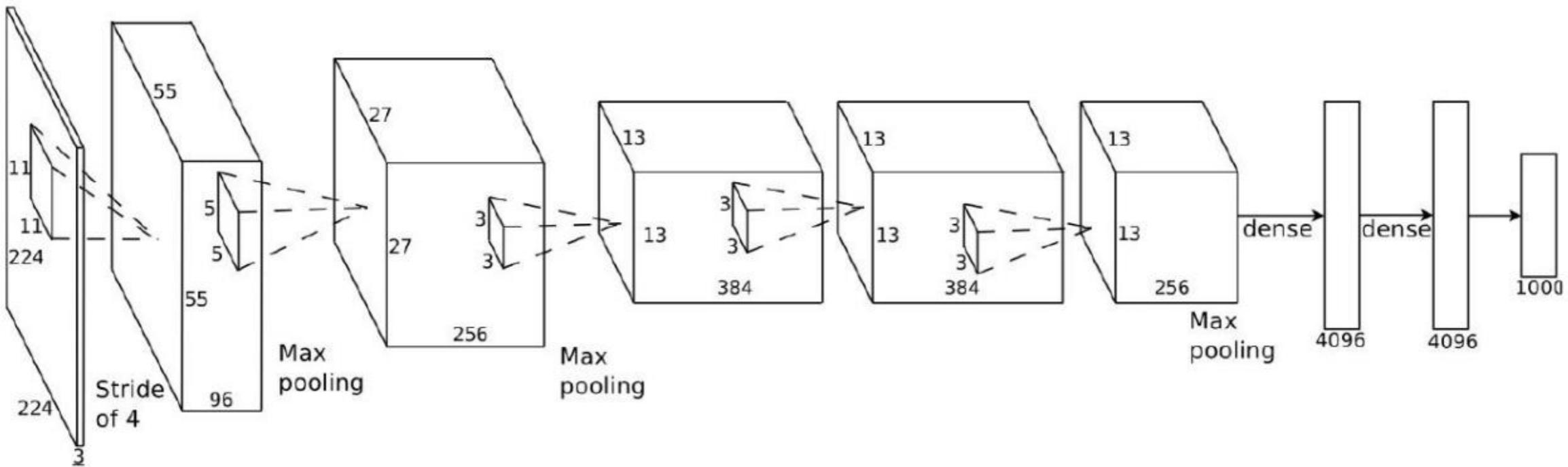
IMAGENET



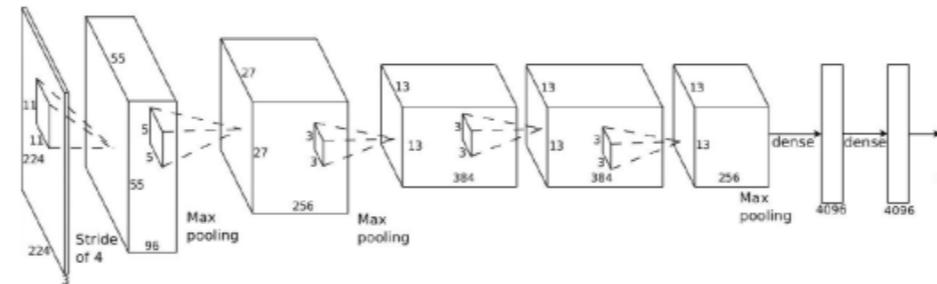
## Comencemos con los datos

- Para realizar el entrenamiento, se utiliza un subconjunto de ImageNet, conocido como ILSVRC-XXXX (donde XXXX es el año).
- 1K categorías, 1.2M de imágenes de entrenamiento (1.2K por categoría).
- 50K imágenes de validación y 150K de test.
- Imágenes RGB de distintas resoluciones, pero escaladas a 256x256 para esta aplicación.
- Métricas de error top-1 y top-5.





Input:  $224 \times 224 \times 3 = 150528$  pixels



- Layer 1:

- nFilters = 96, Size =  $11 \times 11$ , Depth = 3.
- nParams (weights) =  $96 \times 11 \times 11 \times 3 = 34848$ .
- Stride = 4, max-pooling =  $3 \times 3 : 1$ , nNeurons (outputs) =  $55 \times 55 \times 96 = 290400$ .  
(obs: they use 3 zero-padding, then  $(227 - 11)/4 + 1 = 55$ )

- Layer 2:

- nFilters = 256, Size =  $5 \times 5$ , Depth = 96.
- nParams =  $256 \times 5 \times 5 \times 96 = 614400$ .
- Stride = 1, max-pooling =  $3 \times 3 : 2$ , nNeurons =  $27 \times 27 \times 256 = 186624$ .

- Layer 3:

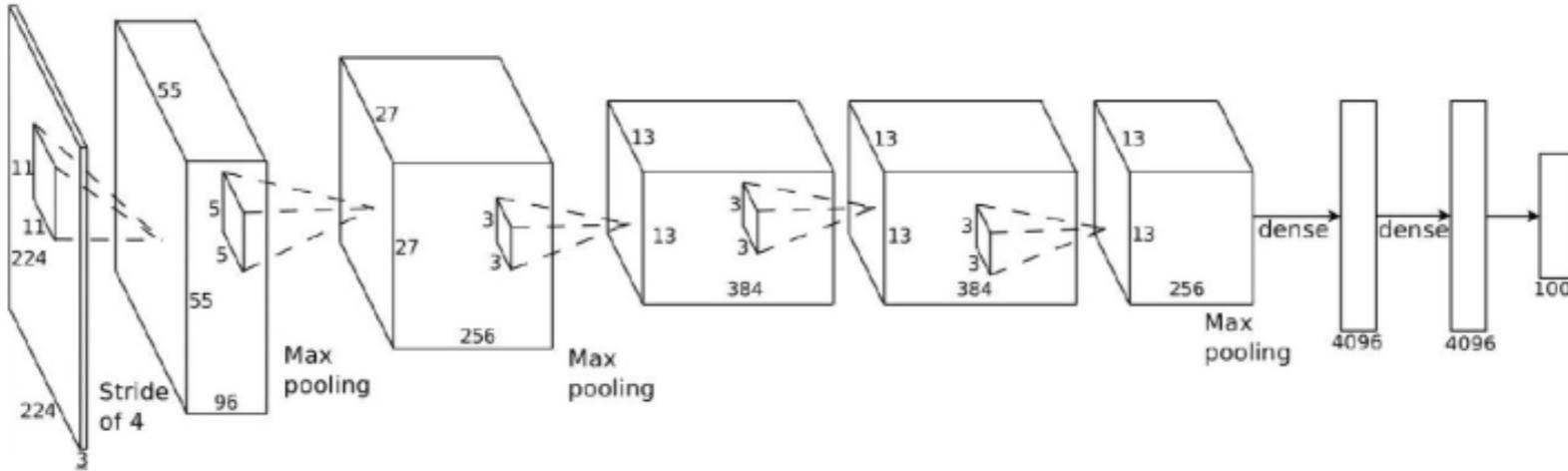
- nFilters = 384, Size =  $3 \times 3$ , Depth = 256.
- nParams =  $384 \times 3 \times 3 \times 256 = 884736$ .
- Stride = 1, max-pooling = none, nNeurons =  $13 \times 13 \times 384 = 64896$ .

- Layer 4:

- nFilters = 384, Size =  $3 \times 3$ , Depth = 384.
- nParams =  $384 \times 3 \times 3 \times 384 = 1327104$ .
- Stride = 1, max-pooling = none, nNeurons =  $13 \times 13 \times 384 = 64896$ .

- Layer 5:

- nFilters = 256, Size =  $3 \times 3$ , Depth = 384.
- nParams =  $256 \times 3 \times 3 \times 384 = 884736$ .
- Stride = 1, max-pooling =  $3 \times 3 : 2$ , nNeurons =  $13 \times 13 \times 256 = 43264$ .



- Layer 6: Fully connected to previous layer

- nParams =  $6 \times 6 \times 256 \times 4096 = 37.748.736$ .
- nNeurons = 4096.

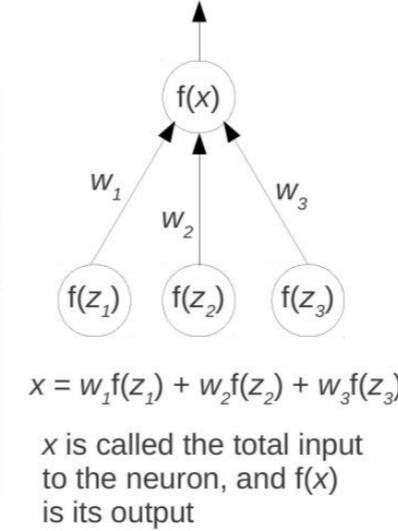
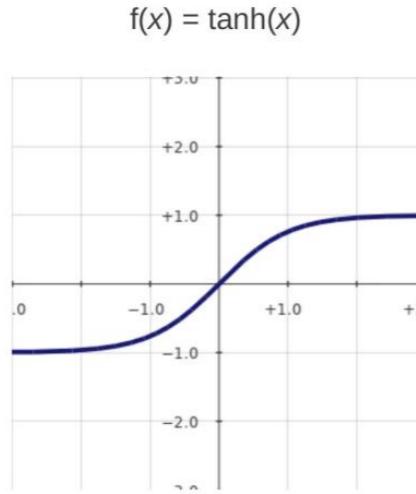
- Layer 7 : Fully connected to previous layer

- nParams =  $4096 \times 4096 = 16.777.216$ .
- nNeurons = 4096.

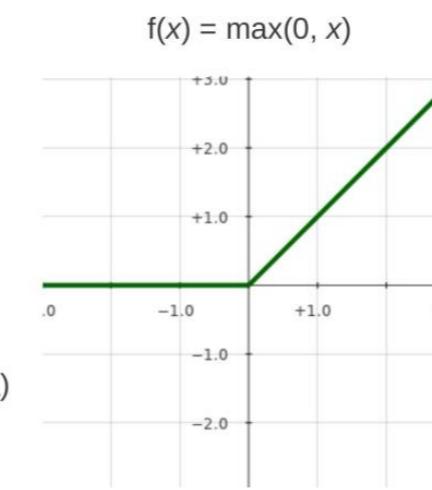
- Layer 8: Fully connected to previous layer

- nParams =  $4096 \times 1000 = 4.096.000$ .
- nNeurons = 1000.

Un aspecto central de su éxito fue el uso la función de activación ReLU



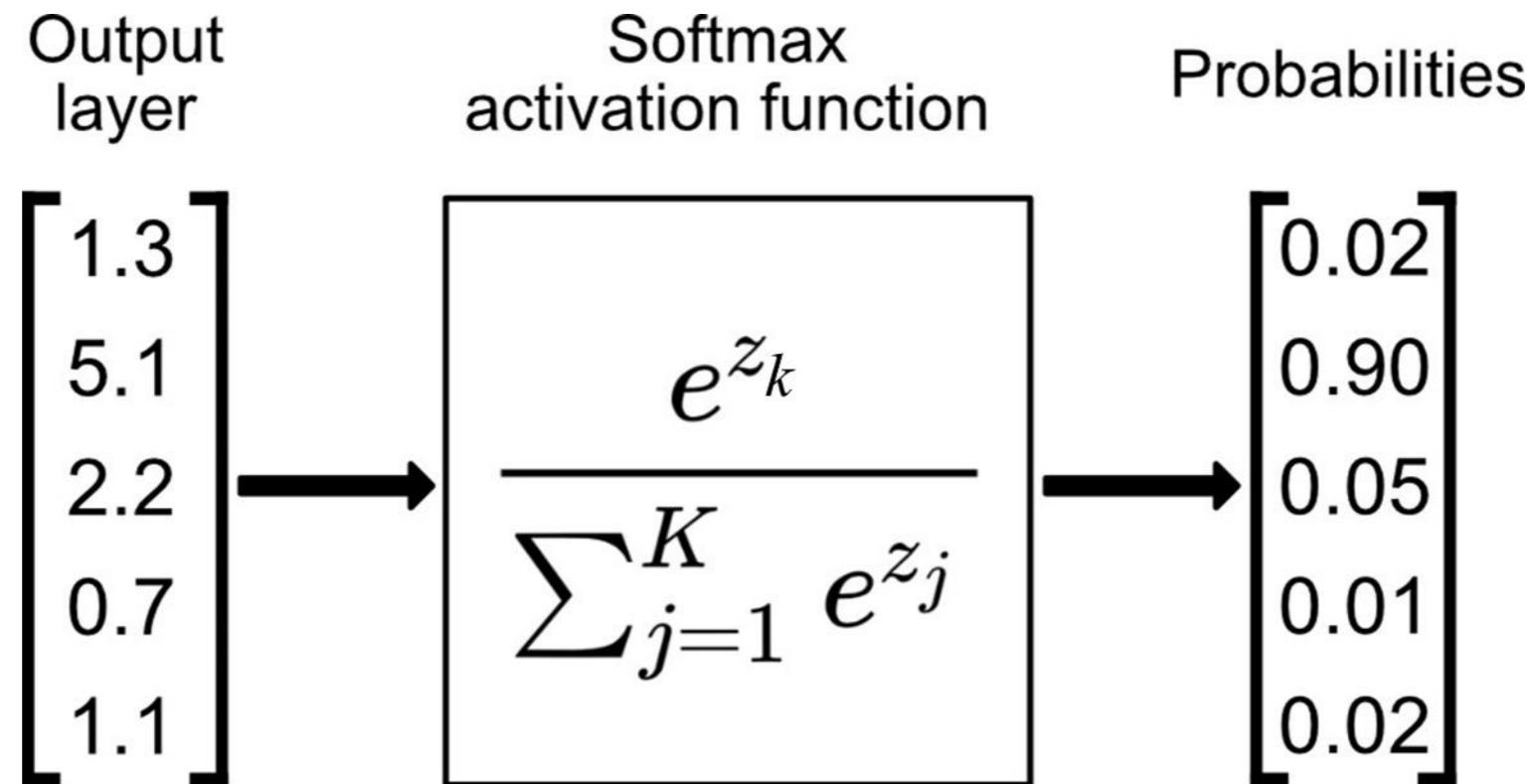
Very bad (slow to train)



Very good (quick to train)

- AlexNet fue uno de los primeros trabajos en cambiar las no linealidades clásicas por ReLU.
- Esto permitió que el entrenamiento fuera varias veces más rápido.

Las salidas de AlexNet son las probabilidades de que la entrada  $x$  corresponda a cada una de las  $K = 1000$  categorías posibles,  $P(Y = k | X = x)$ . Esto se logra utilizando la función de activación *softmax* luego de la última capa.



Como función de perdida, se utilizó la función Cross Entropy, que combinada con la función *softmax*, genera una expresión bastante simple y compacta

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

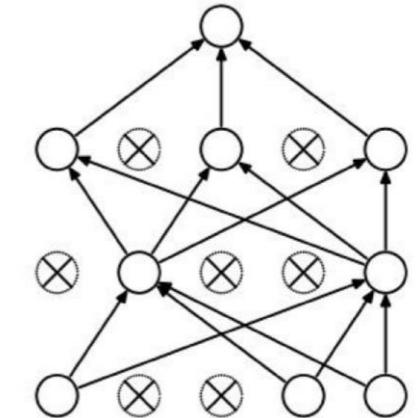
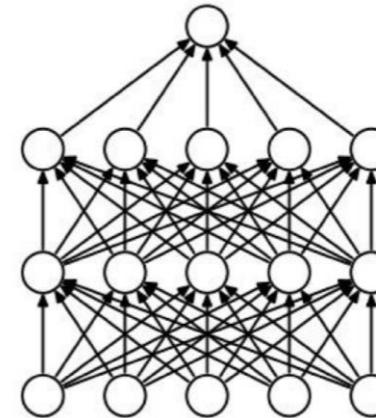
$$L_i = -\log P(Y = y_i | X = x_i)$$

---

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

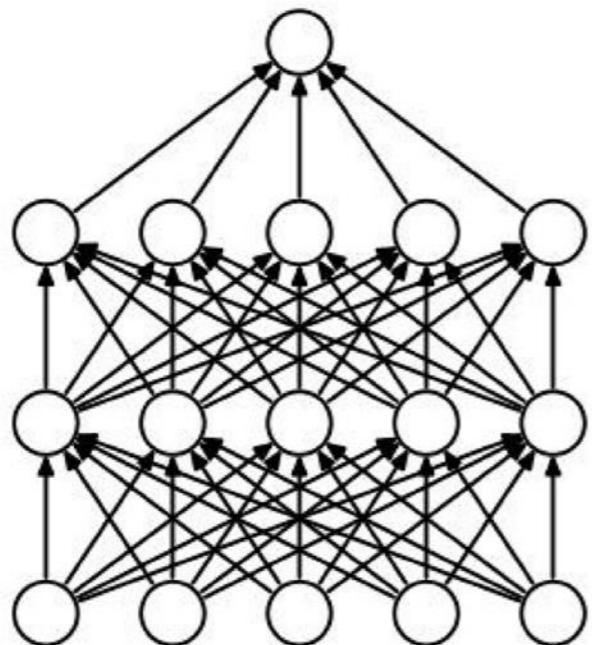
## ¿Cómo controlar el sobreajuste en redes con gran cantidad de parámetros?

- Capas densas del final son igualmente propensas a sobreajustarse.
- Para evitar esto, se introdujo en 2012 una técnica llamada **Dropout**, aplicada a las capas densas.
- **Dropout** “apaga” aleatoriamente neuronas en estas capas durante el entrenamiento.
- El efecto neto es que las neuronas no pueden “confiar” en las otras, por lo que aprenden representaciones más robustas.

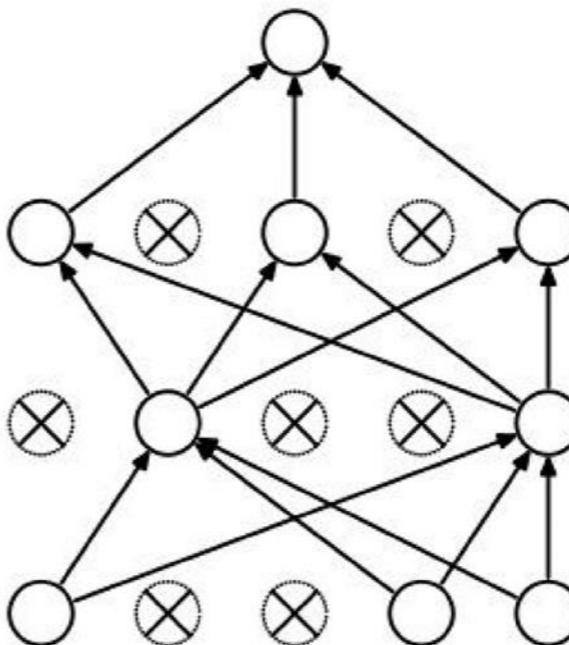


# Regularization: Dropout

“randomly set some neurons to zero in the forward pass”

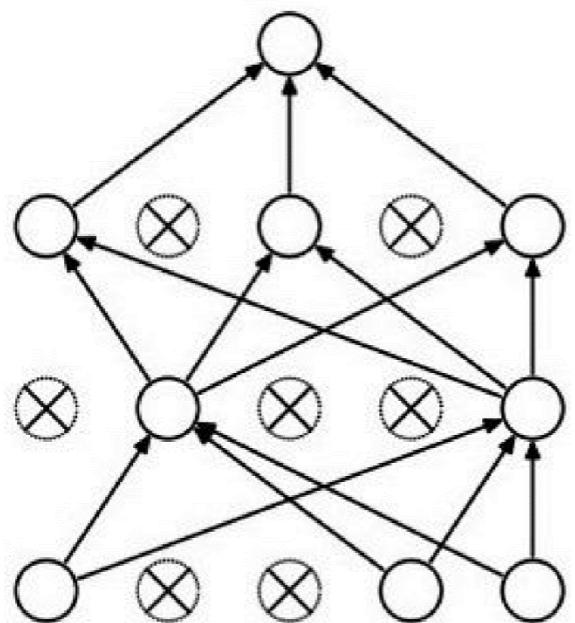


(a) Standard Neural Net

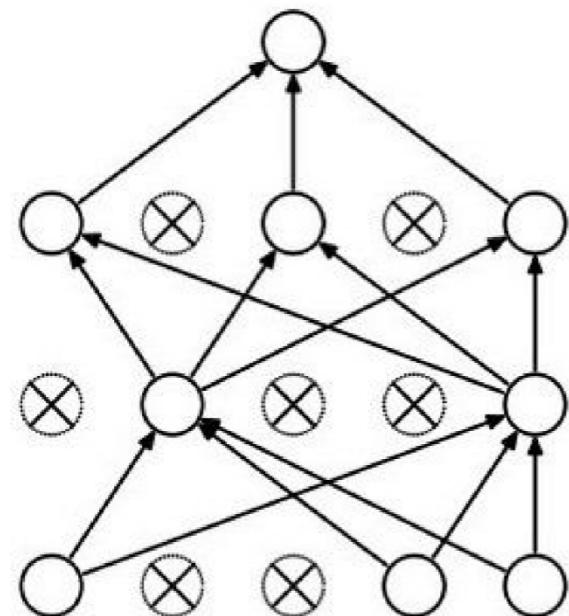


(b) After applying dropout.

Waaaait a second...  
How could this possibly be a good idea?



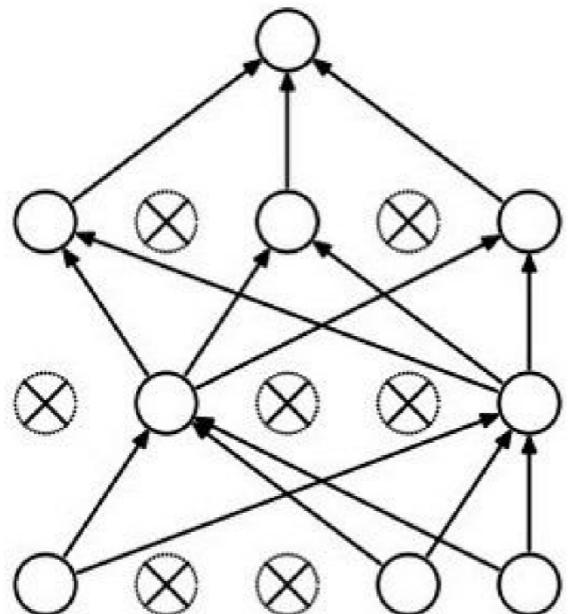
Waaaait a second...  
How could this possibly be a good idea?



Forces the network to have a redundant representation.



Waaaait a second...  
How could this possibly be a good idea?



Another interpretation:

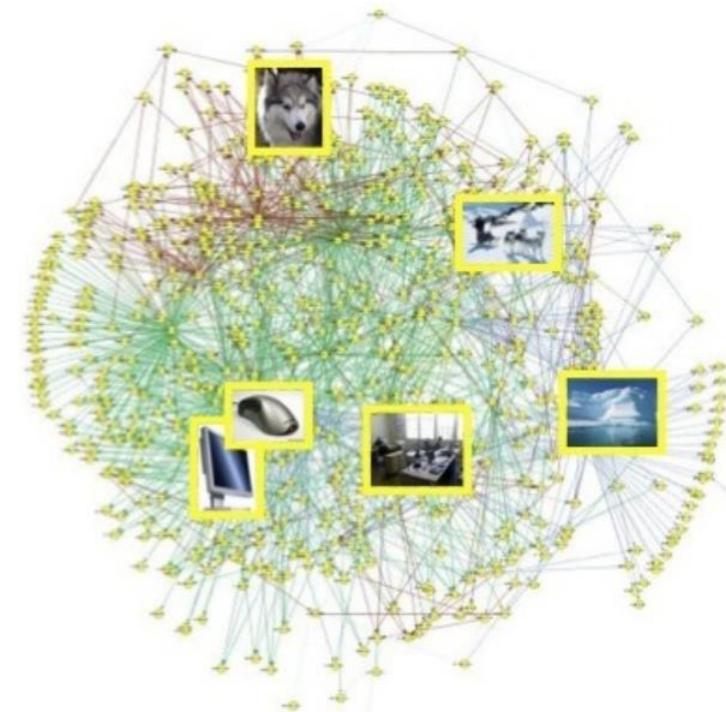
Dropout is training a large ensemble  
of models (that share parameters).

Each binary mask is one model, gets  
trained on only ~one datapoint.

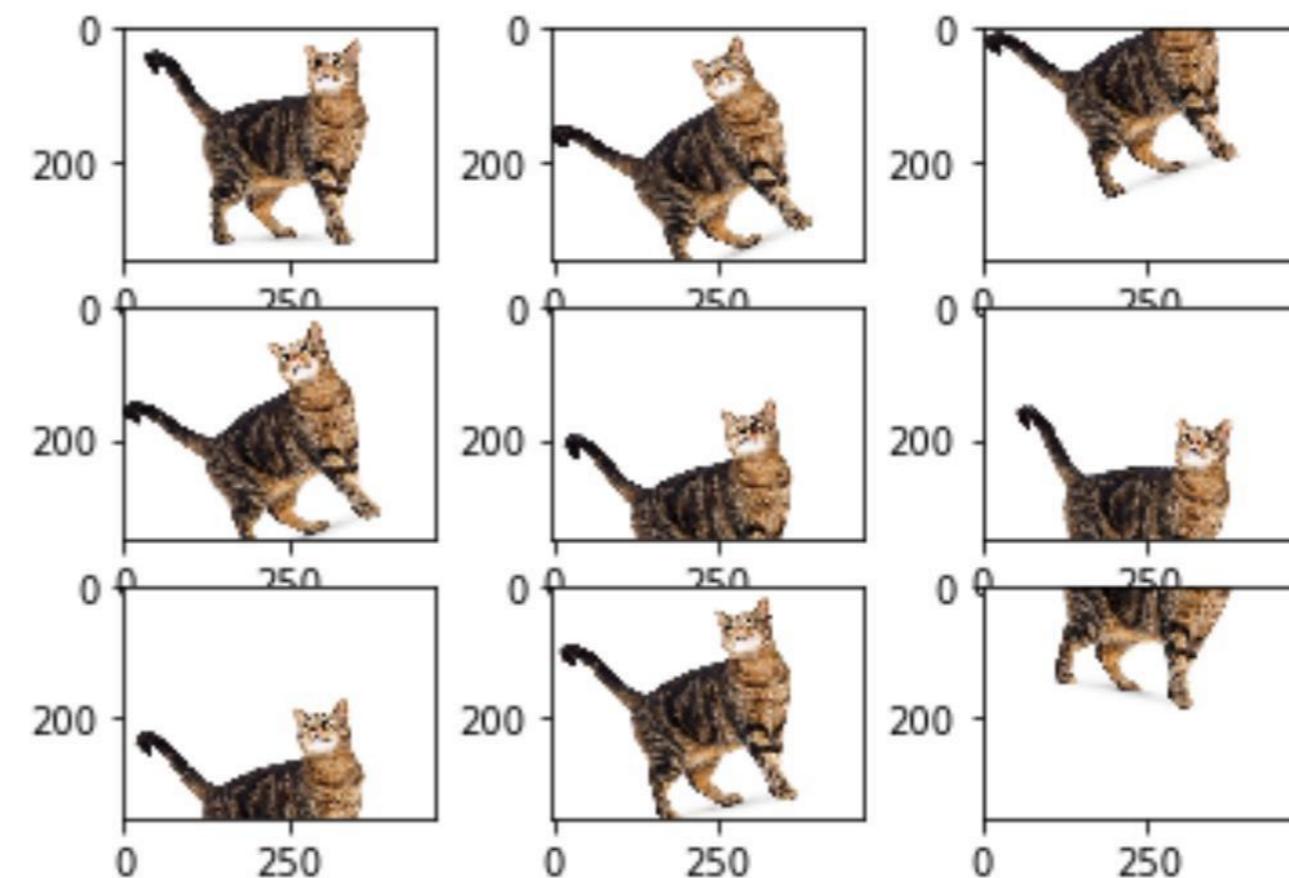
Otro aporte de AlexNet fue popularizar técnicas de **aumento de datos** para controlar el sobreajuste

- Si bien el set de datos es grande, el tamaño de la red obliga a aumentarlo aún más.
- Para esto se utilizaron técnicas de **aumento de datos**
- Por cada imagen se tomaron 5 subimágenes de 224x224 (4 esquinas y centro).
- Este proceso también se repite para el reflejo de cada imagen.
- Con esto, lograron aumentar 10 veces el tamaño efectivo del set de datos.
- Finalmente, a cada imagen se le resta la media de cada canal de color

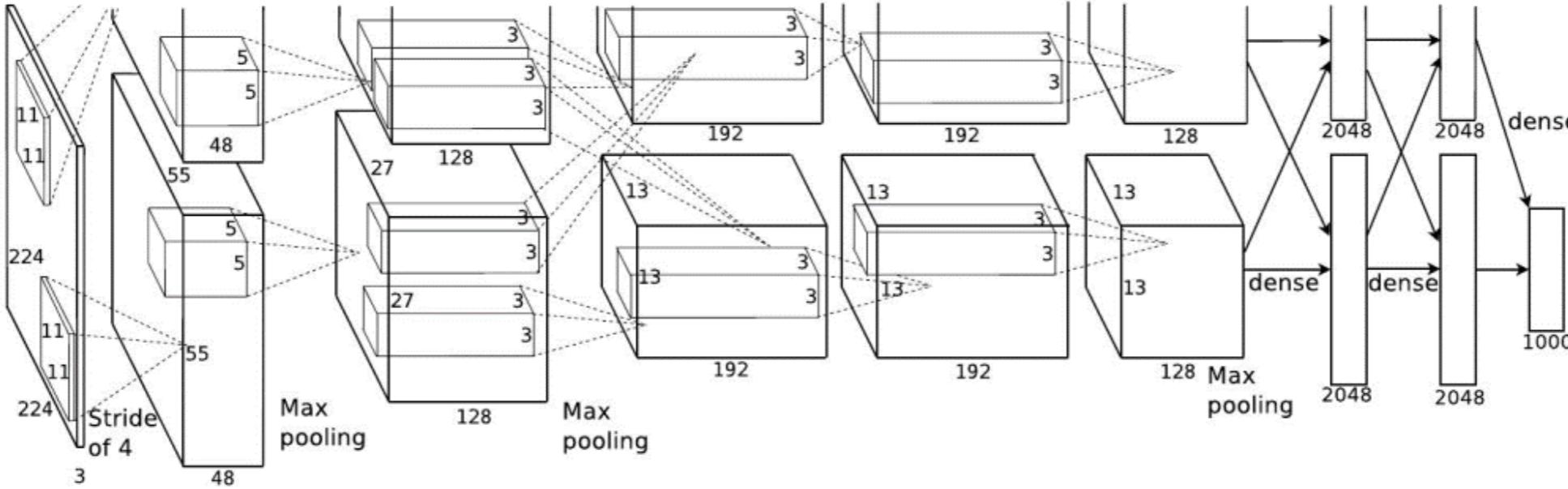
IMAGENET



Aumento de datos puede verse como la otra cara de la moneda de la Navaja de Occam => aumentamos la complejidad del problema, en vez de reducir la del modelo

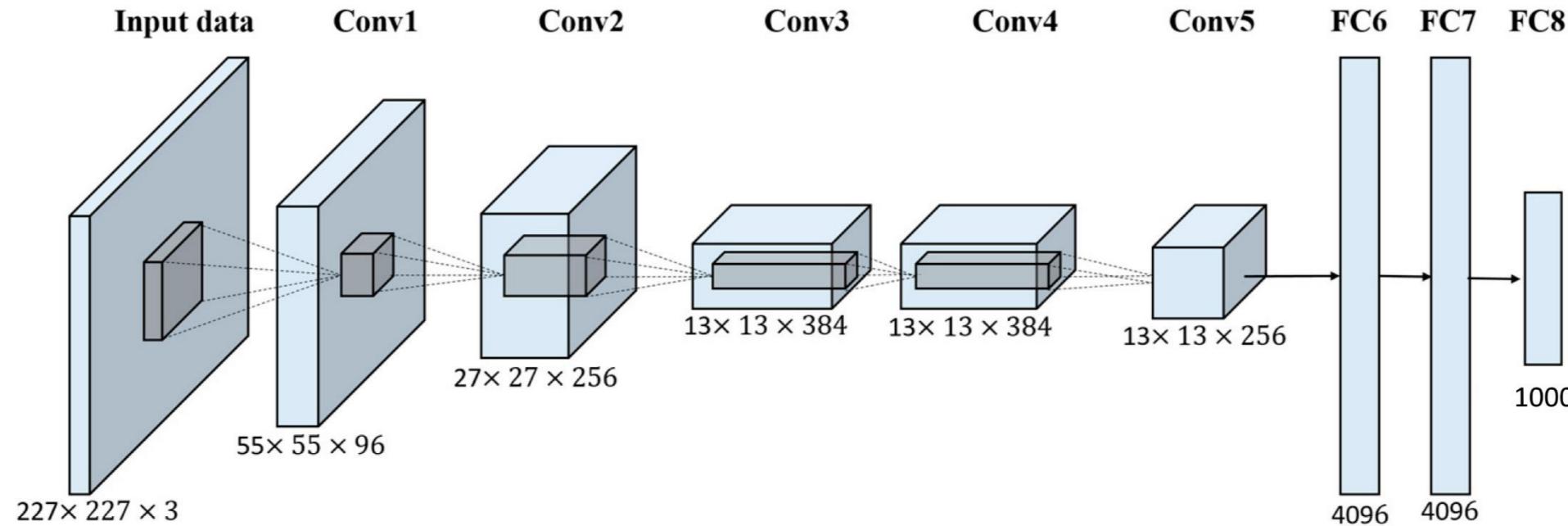


AlexNet fue entrenada utilizando “solo” 2 GPUs



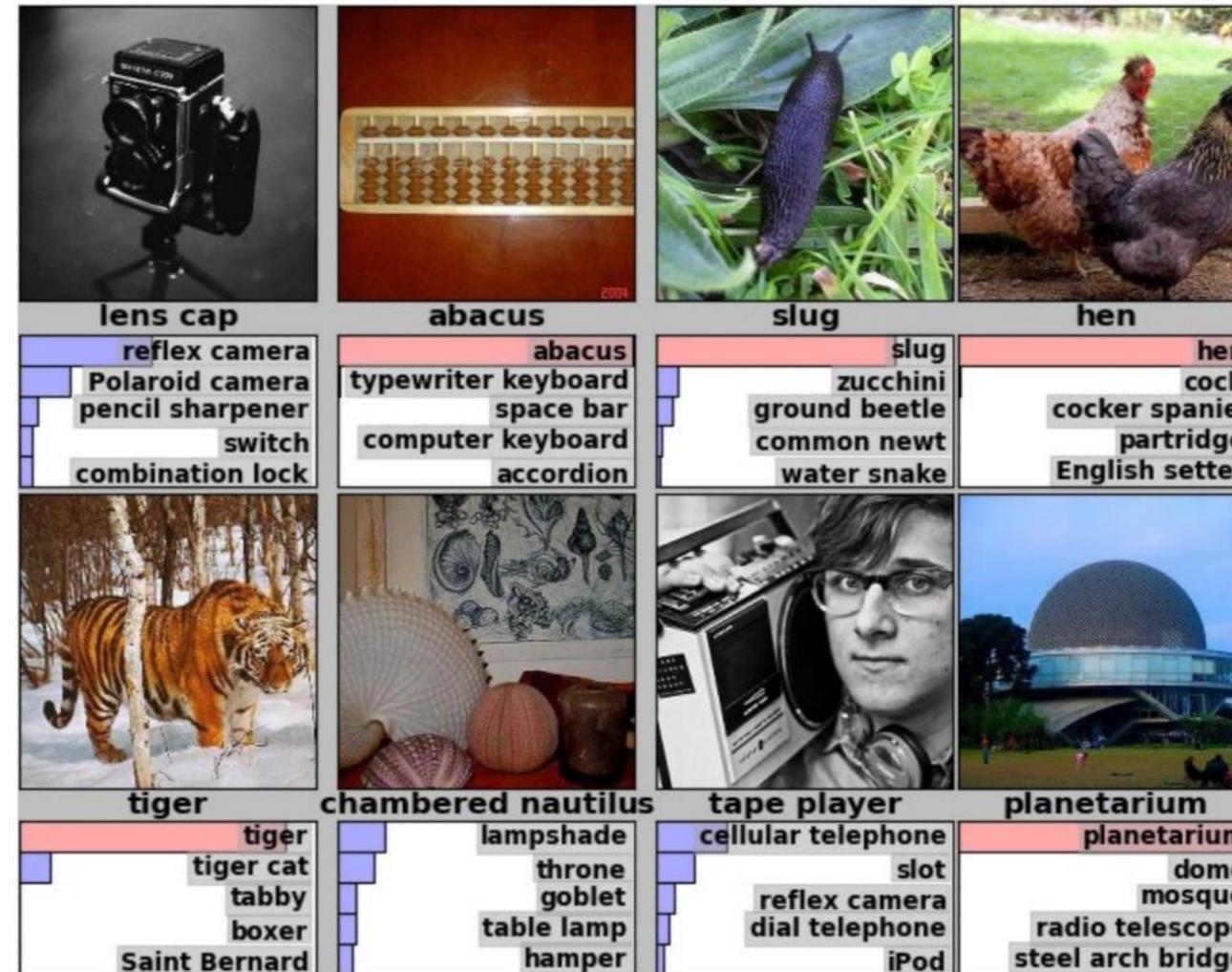
- Entrenada utilizando SGD en dos Nvidia GTX 580 con 3GB de VRAM durante una semana.
- Algunos números aproximados para cuantificar el tamaño:
  - $\approx 650K$  neuronas
  - $\approx 60M$  de parámetros
  - $\approx 630M$  de conexiones

La combinación de todo esto generó resultados sorprendentes para la época

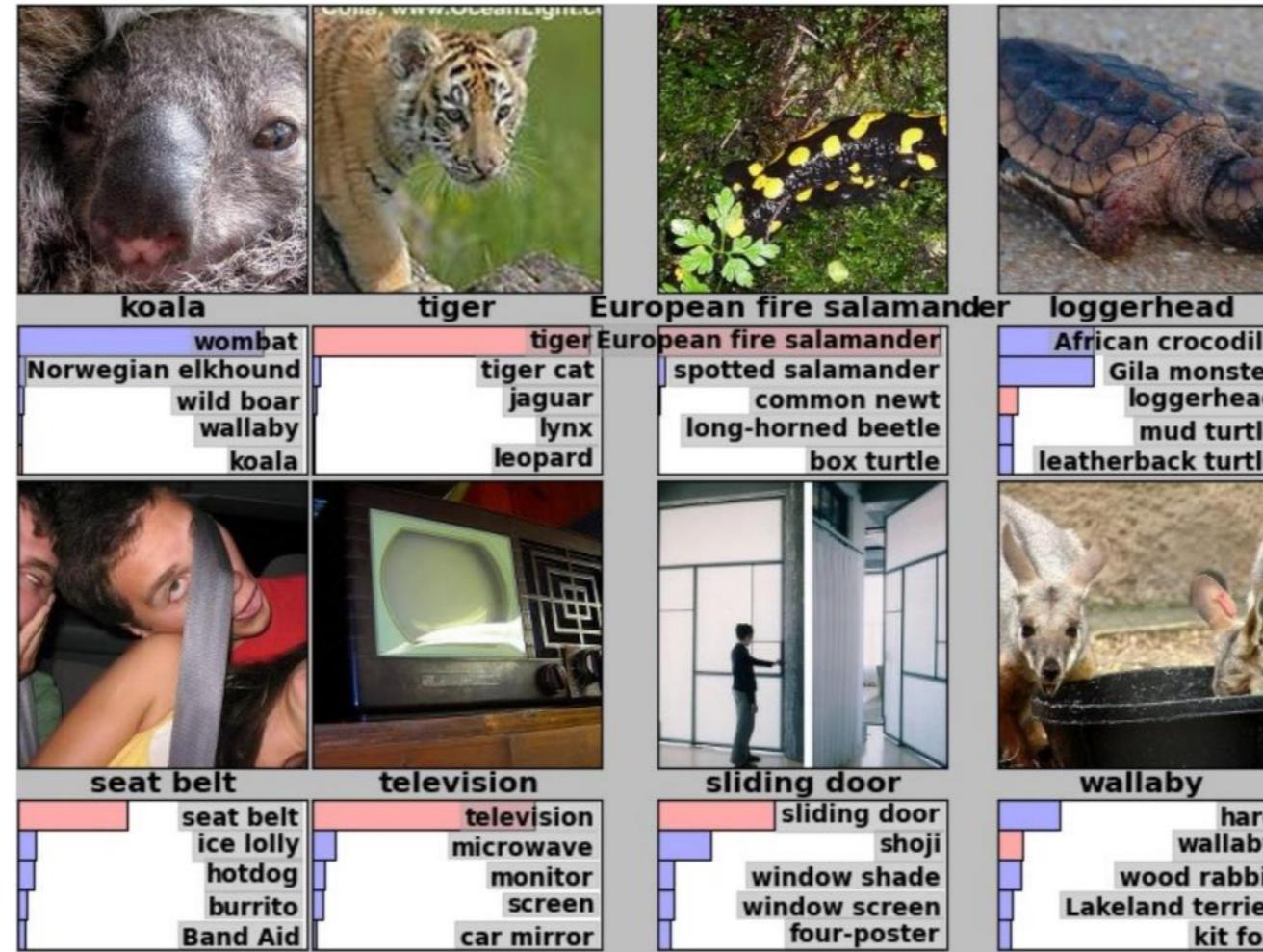


- Error top-5: 15.3%
- Error top-1: 26.2%
- Error humano: aprox. 8%

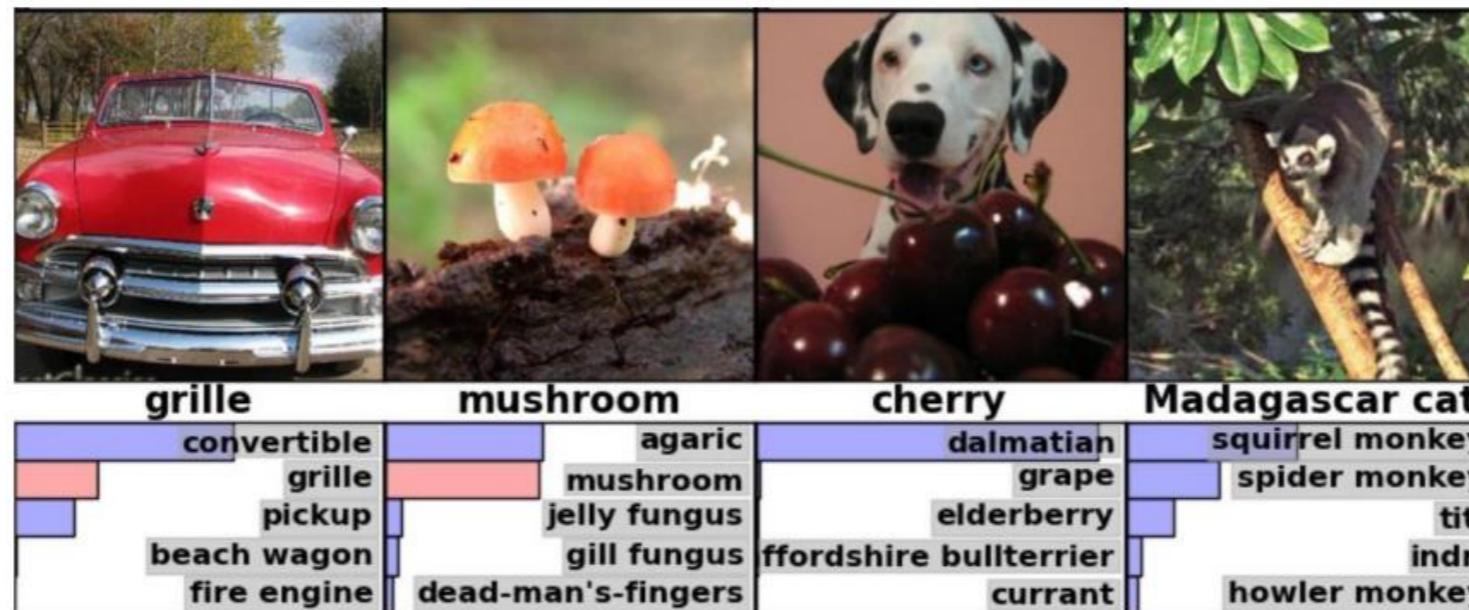
La combinación de todo esto generó resultados sorprendentes para la época



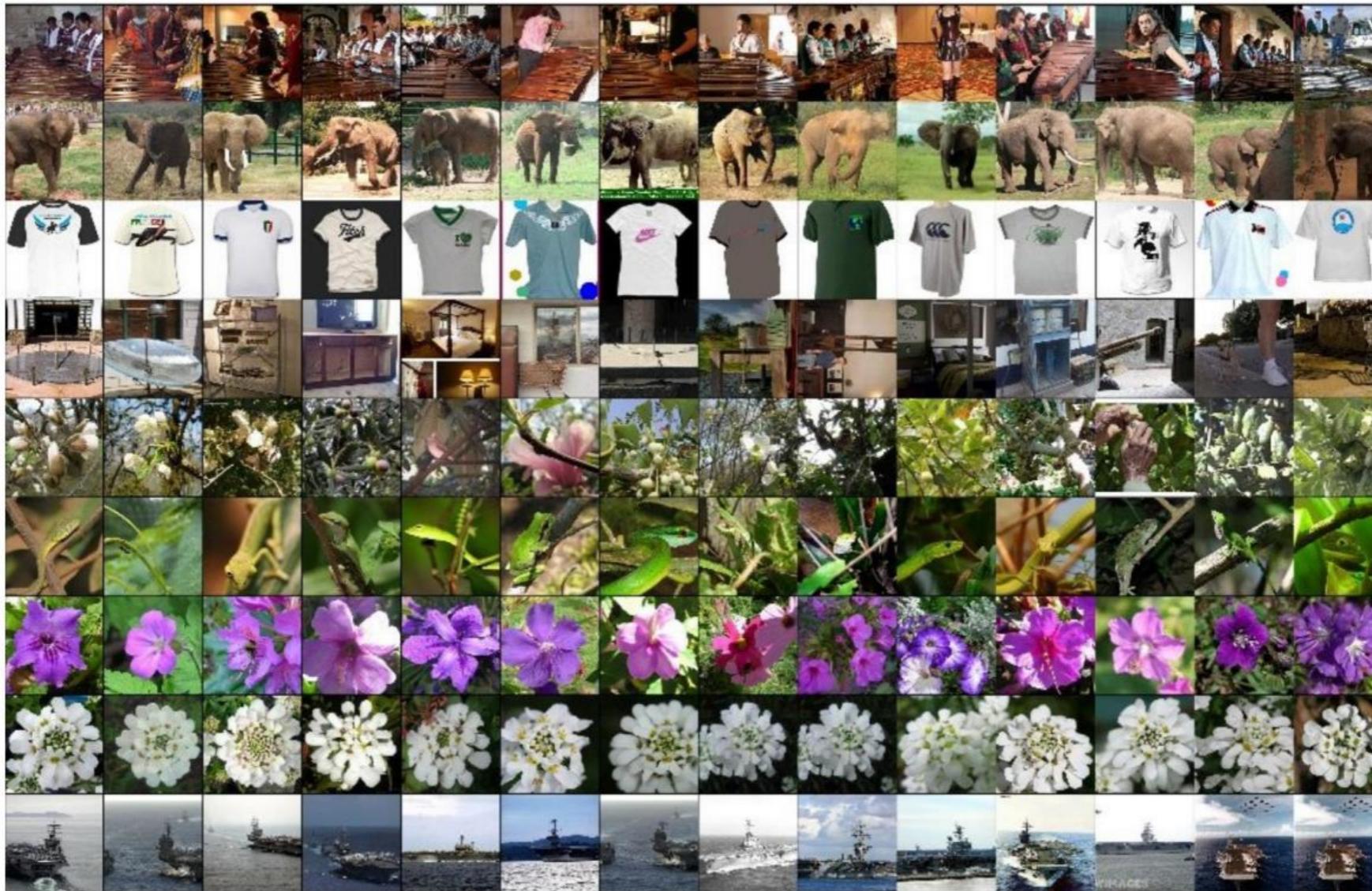
La combinación de todo esto generó resultados sorprendentes para la época



La combinación de todo esto generó resultados sorprendentes para la época



La combinación de todo esto generó resultados sorprendentes para la época



Rendimiento ha continuado mejorando



(a) Siberian husky



(b) Eskimo dog

- En la actualidad, las arquitecturas para reconocimiento visual son bastante más profundas (cientos de capas).
- Mejores métodos tienen alrededor de 89% top-1 y 99% top-5.
- Algunas muestras con código pueden verse en  
<https://paperswithcode.com/sota/image-classification-on-imagenet>

Pontificia Universidad Católica de Chile  
Escuela de Ingeniería  
Departamento de Ciencia de la Computación



# IIC2613 - Inteligencia Artificial

Entrenamiento de Redes Neuronales

Hans Löbel  
Dpto. Ingeniería de Transporte y Logística  
Dpto. Ciencia de la Computación