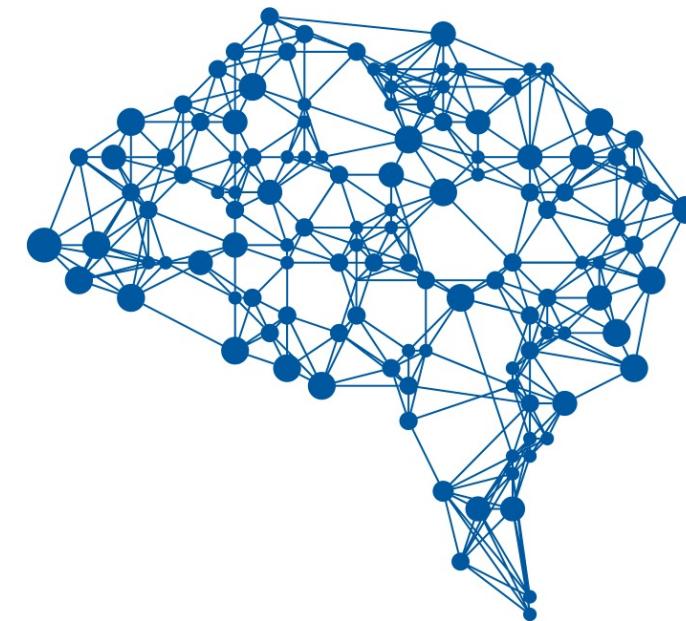

Clase 2.A

Perceptrón Multicapa y Backpropagation

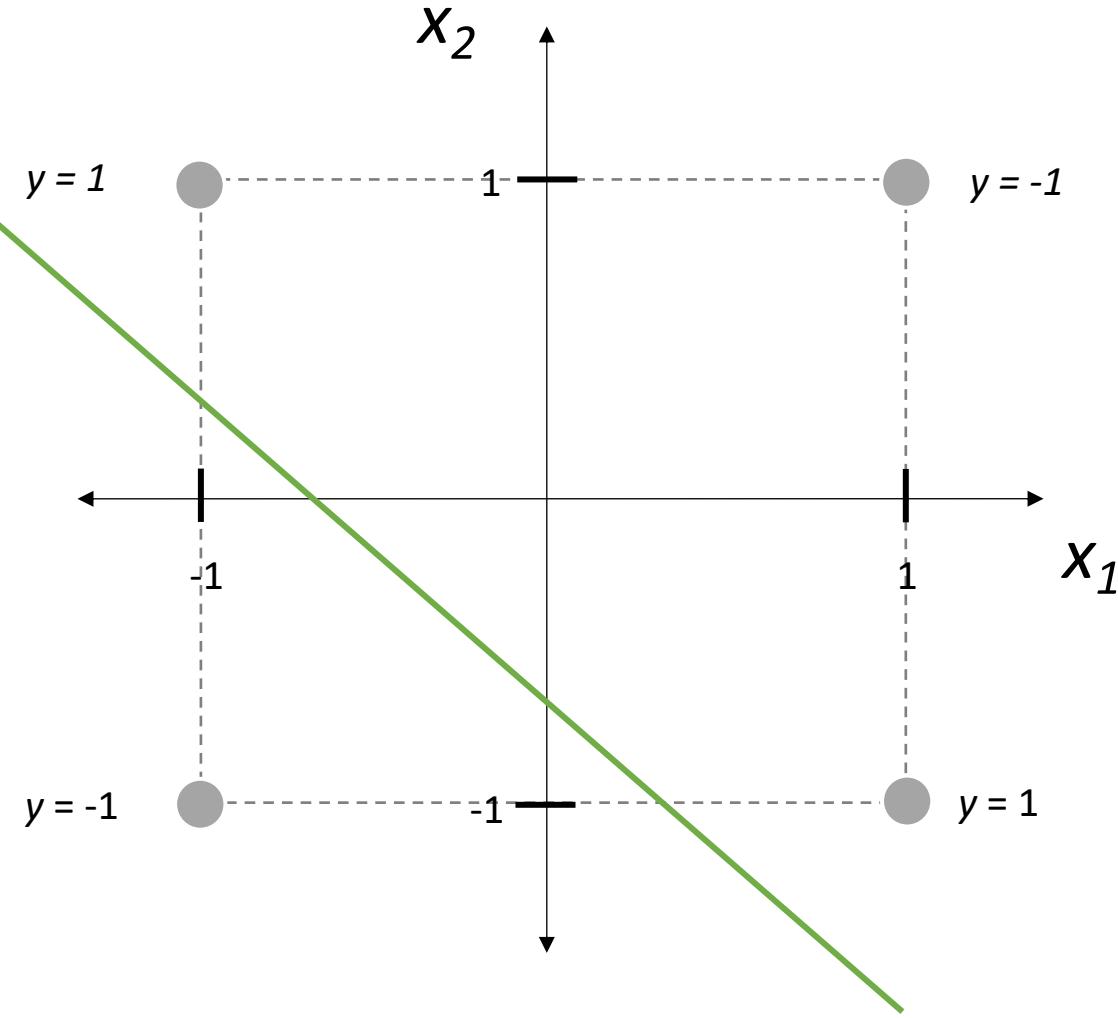
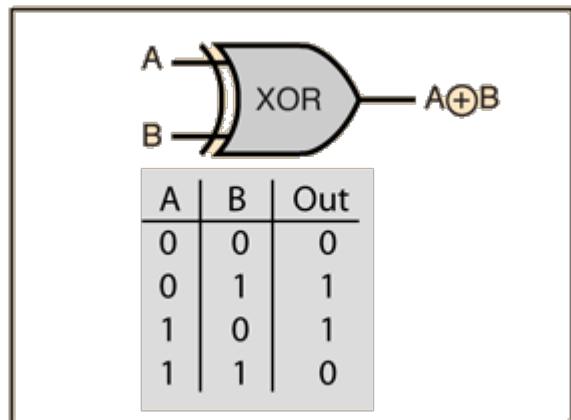
Enzo Ferrante

 eferrante@sinc.unl.edu.ar

 @enzoferante

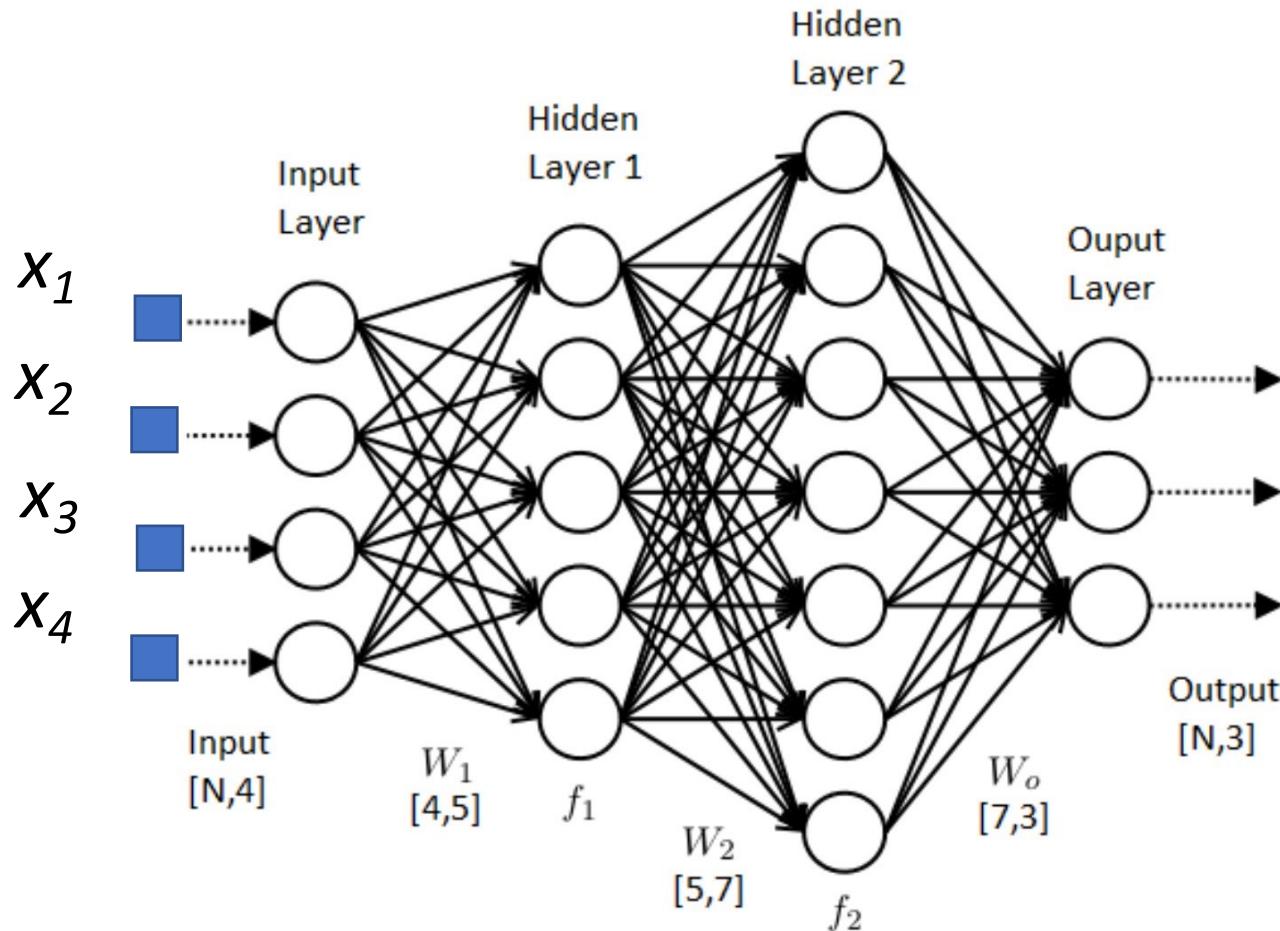


El perceptrón simple puede modelar el XOR?



Perceptrón Multicapa

Multi layer perceptron (MLP)



Método del gradiente para MLP

1. Inicializar los pesos \mathbf{W} aleatoriamente

2. Dado el dataset $\mathcal{D} = \{(\mathbf{x}, d)_n\} = \{(\mathbf{x}, d)_1, \dots, (\mathbf{x}, d)_N\}$

Actualizar en
la dirección opuesta
al gradiente de la
pérdida promedio

3. Volver a 2 hasta satisfacer algún criterio de convergencia

$$\mathcal{L}(\mathcal{D}; \mathbf{W}) = \frac{1}{N} \sum_{n=1}^N \mathcal{L}((\mathbf{x}, d)_n; \mathbf{W})$$

$$\mathbf{W} = \mathbf{W} - \delta \nabla_{\mathbf{W}} \mathcal{L}(\mathcal{D}; \mathbf{W})$$

Matriz con un vector de pesos \mathbf{w}_i por capa



Cálculo del gradiente sobre el
dataset de entrenamiento
completo

Variantes del método del gradiente

- **Gradiente descendiente clásico**
- **Gradiente descendiente estocástico**
 - **Gradiente descendiente por mini-batches**

Método del gradiente descendente por mini-batches

1. Inicializar los pesos \mathbf{w} aleatoriamente

2. Dado el dataset $\mathcal{D} = \{(\mathbf{x}, d)_n\} = \{(\mathbf{x}, d)_1, \dots, (\mathbf{x}, d)_N\}$

3. Por cada minibatch $\mathcal{D}_M = \{(\mathbf{x}, d)_m\} = \{(\mathbf{x}, d)_1, \dots, (\mathbf{x}, d)_M\}, M < N$

Actualizar en
la dirección opuesta
al gradiente de la
pérdida promedio

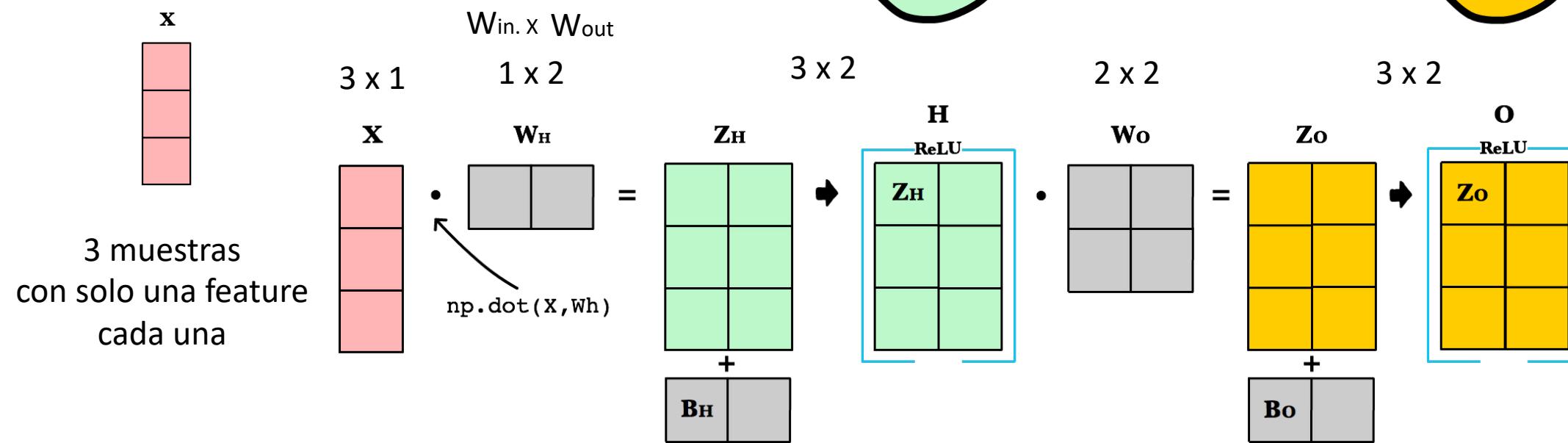
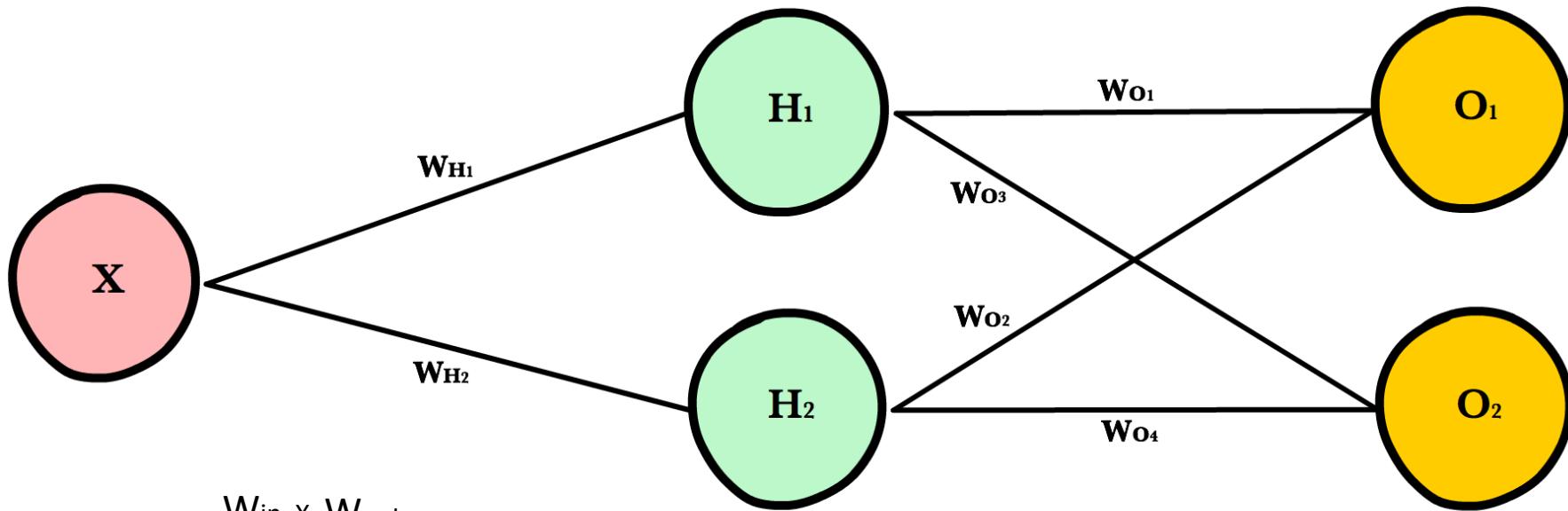
$$\mathbf{W} = \mathbf{W} - \delta \nabla_{\mathbf{W}} \mathcal{L}(\mathcal{D}_M; \mathbf{W})$$

4. Volver a 2 hasta satisfacer algún criterio de convergencia

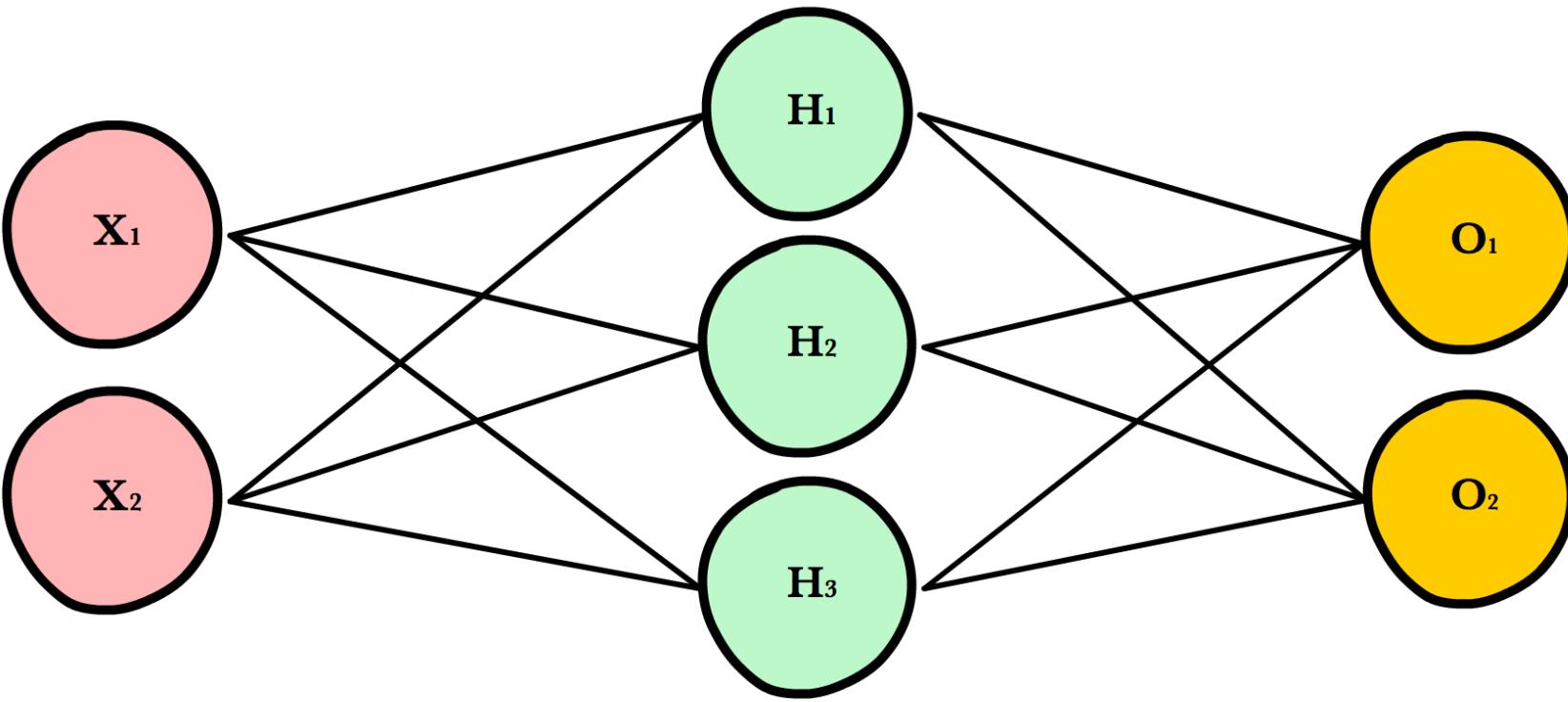
$$\mathcal{L}(\mathcal{D}_M; \mathbf{W}) = \frac{1}{M} \sum_{m=1}^M \mathcal{L}((\mathbf{x}, d)_m; \mathbf{W})$$

Cómputo del gradiente sobre el
mini-batch de tamaño $M < N$

Perceptrón Multicapa



Perceptrón Multicapa



3 muestras
con 2 features
de entrada cada una

X_1	X_2
1	1
1	0
0	1
0	0

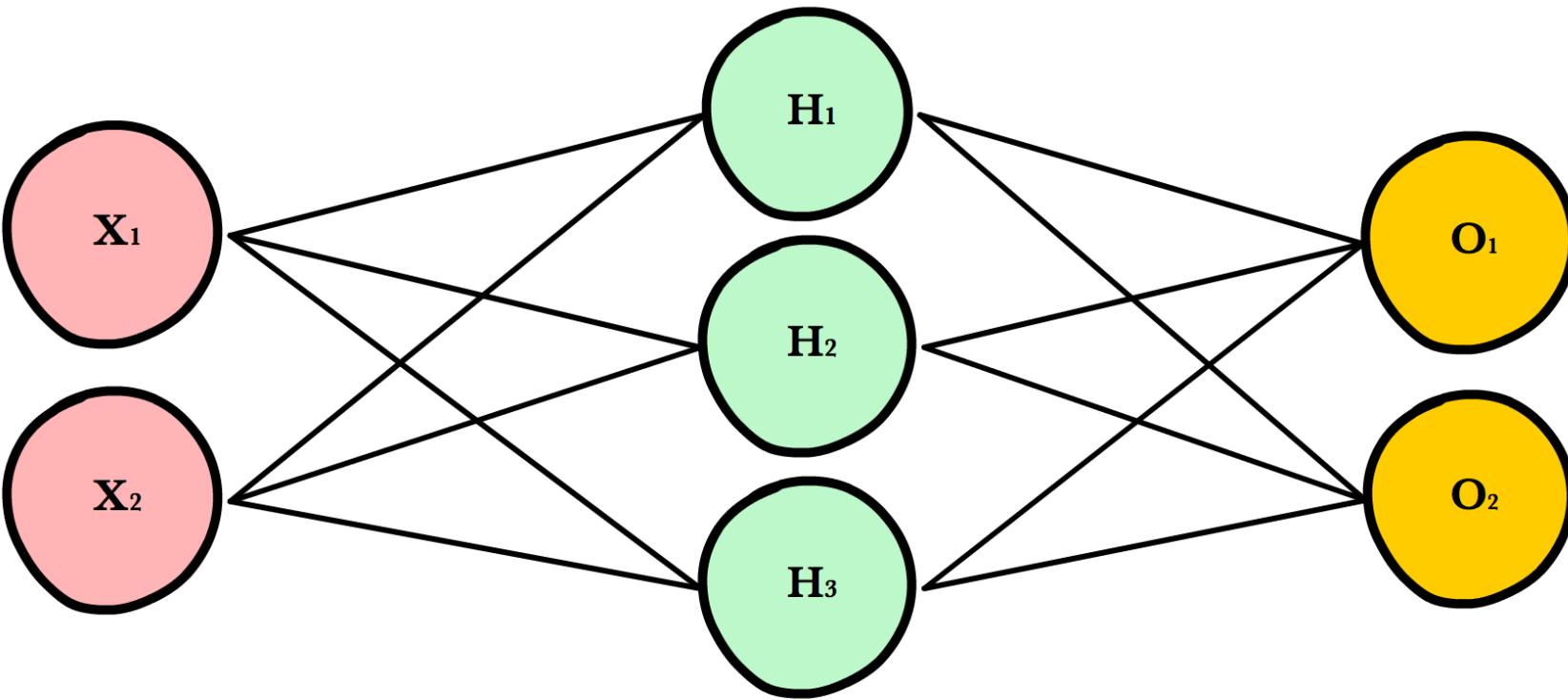
W_H
1 1 1
1 1 1

H_1	H_2	H_3
1 1 1	1 1 1	1 1 1
1 1 1	1 1 1	1 1 1
1 1 1	1 1 1	1 1 1
1 1 1	1 1 1	1 1 1

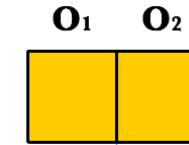
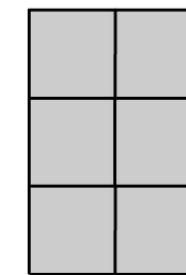
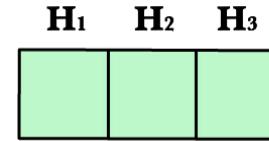
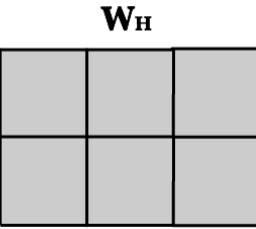
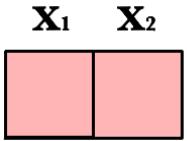
W_O
1 1
1 1

O_1	O_2
1	1
1	1
1	1
1	1

Perceptrón Multicapa



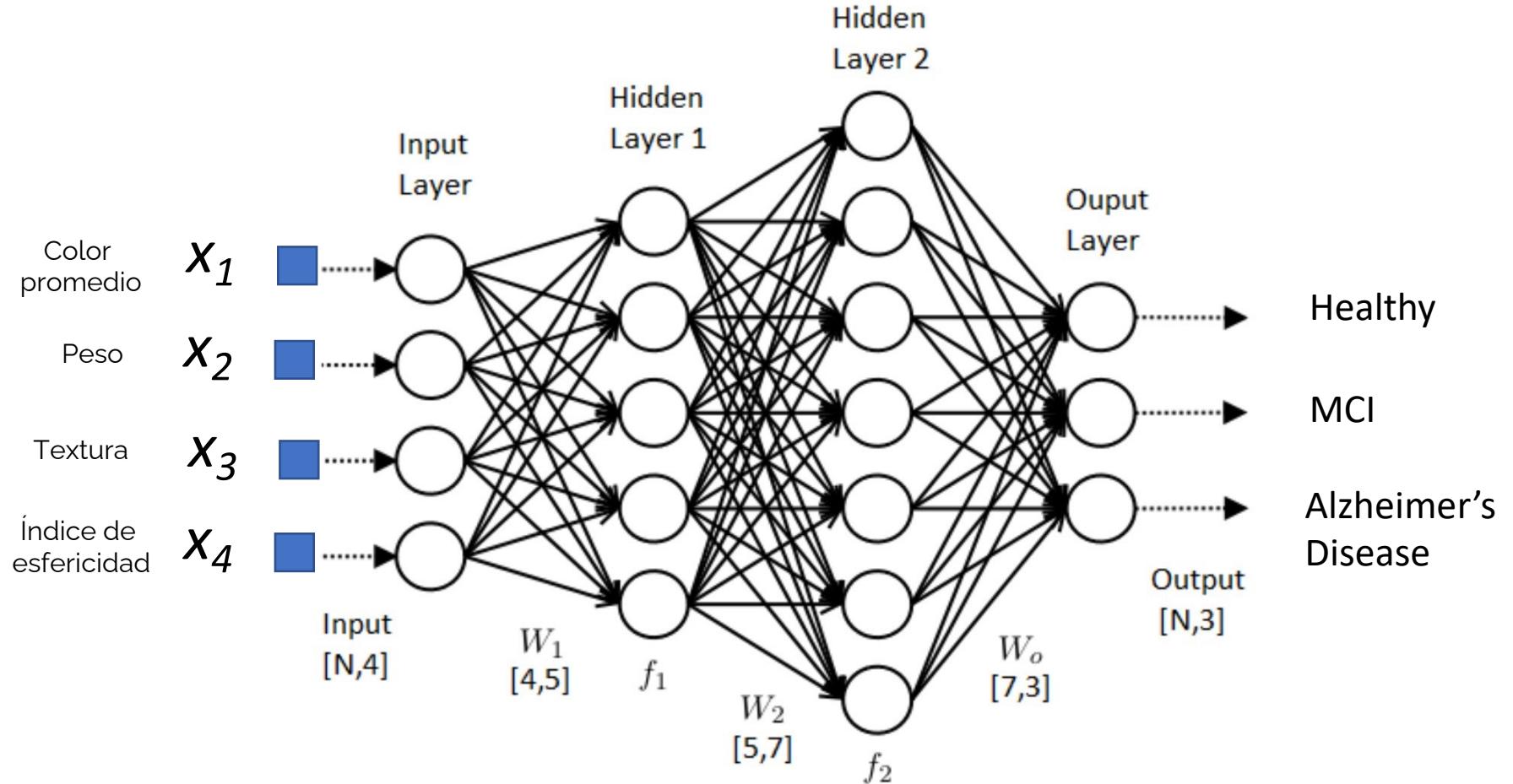
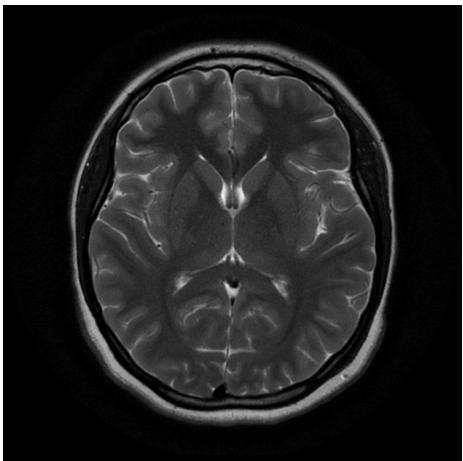
1 muestra
con 2 features
de entrada



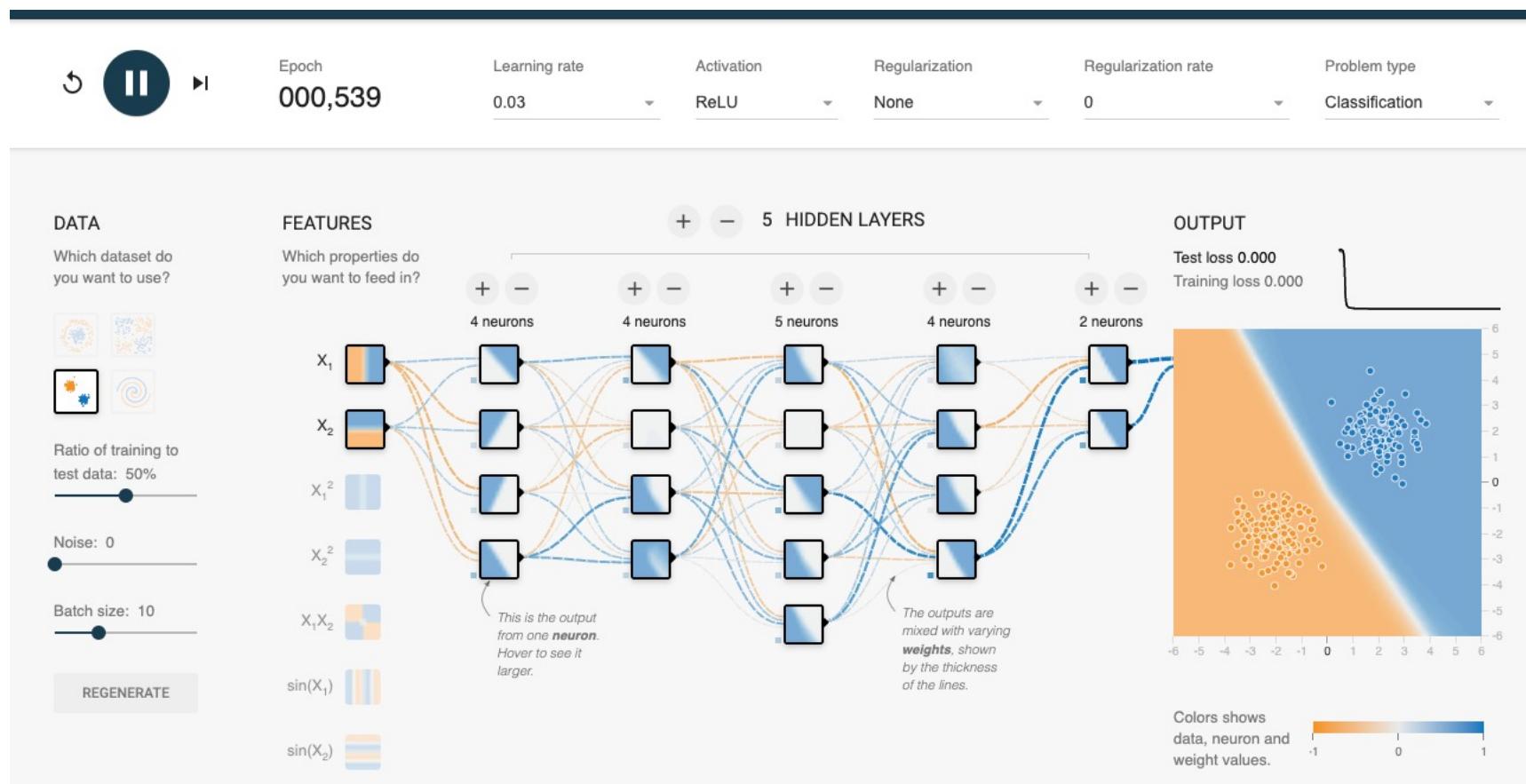
El tamaño de las matrices
 W es independiente
de la cantidad de
elementos en el batch!

Perceptrón Multicapa

Ejemplo: Problema de clasificación con 4 features de entrada y 3 clases de salida



Perceptrón Multicapa

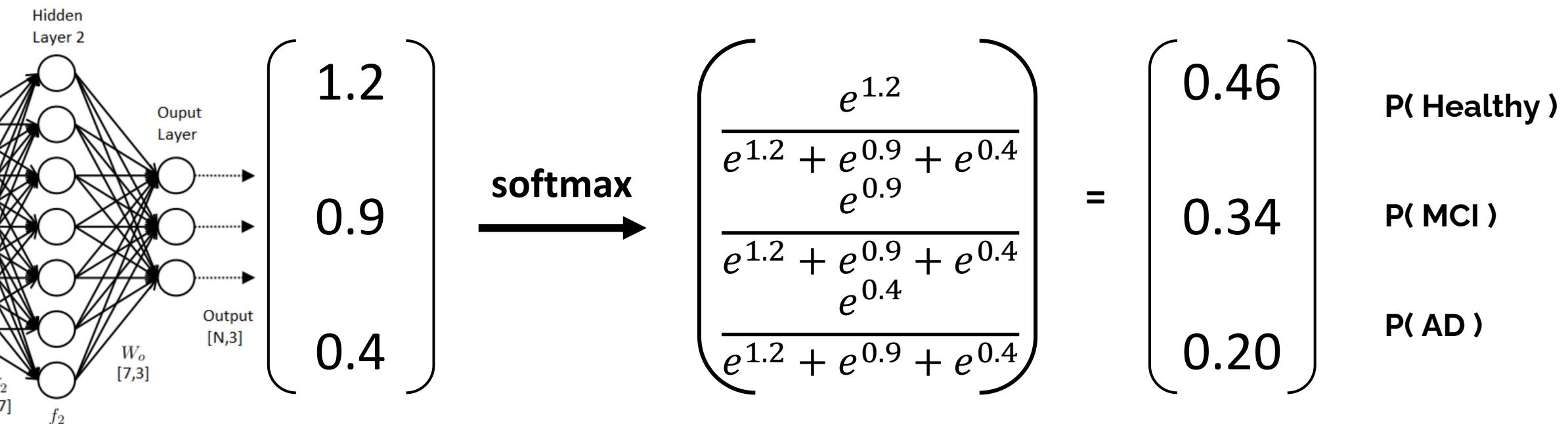


<https://playground.tensorflow.org/>

Softmax como función activación de salida

Permite obtener una "interpretación probabilística" de la salida

$$\text{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{c=1}^C e^{z_c}}$$



Función de pérdida para problemas de clasificación

Entropía Cruzada entre dos distribuciones de probabilidad p y q

$$CE(p, q) = - \sum_x p(x) \log q(x)$$

Distribución Ground Truth
↓
Distribución Estimada

Categoría	Etiqueta
Manzana	1
Banana	2
Naranja	3

Notación One hot		
1	0	0
0	1	0
0	0	1

Función de pérdida para problemas de clasificación

Entropía Cruzada Binaria

Ground Truth

$$-p \log (q) - (1-p) \log(1-q)$$

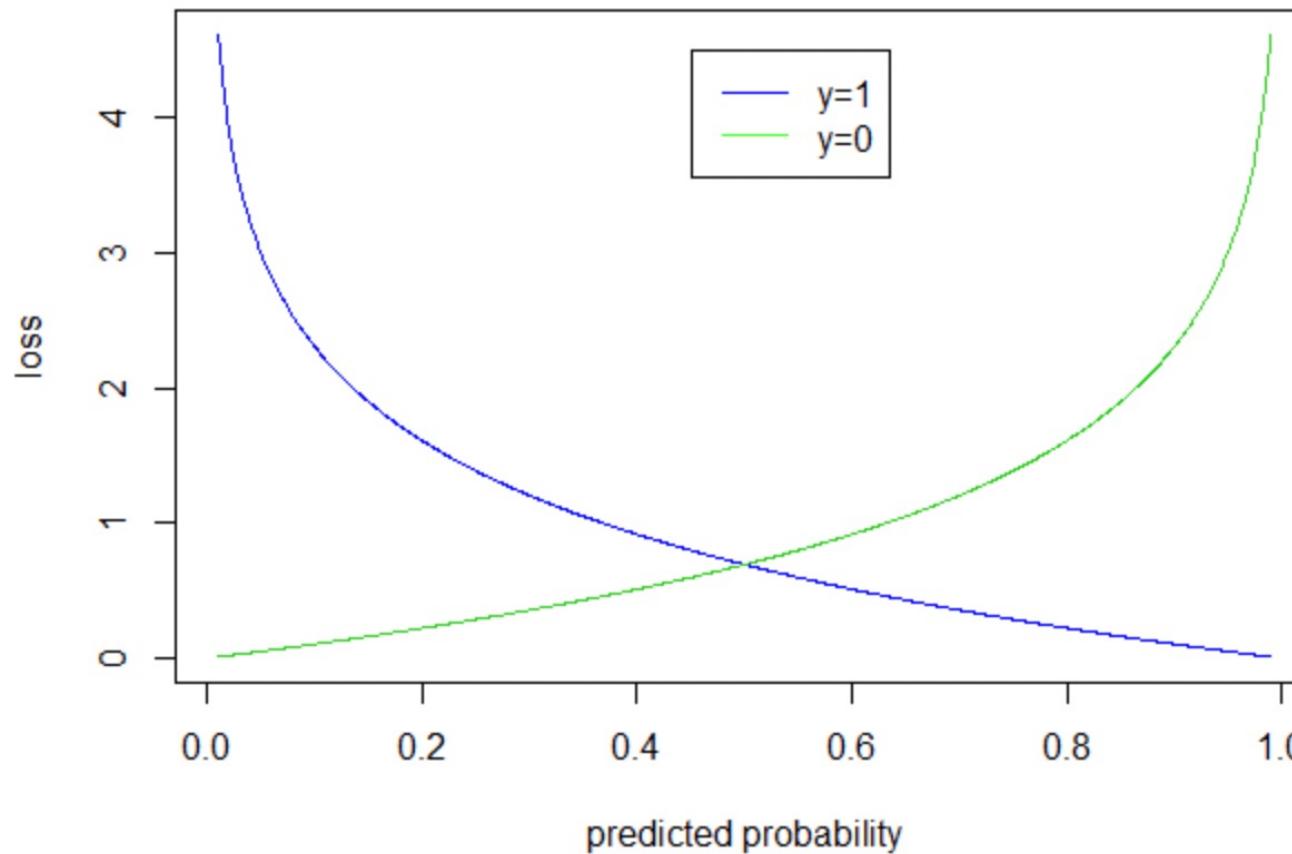
Estimada

Función de pérdida para problemas de clasificación

Entropía Cruzada Binaria

Ground Truth Estimada

$$-y \log (\hat{y}) - (1-y) \log(1-\hat{y})$$



Función de pérdida para problemas de clasificación

Entropía Cruzada entre dos distribuciones de probabilidad p y q

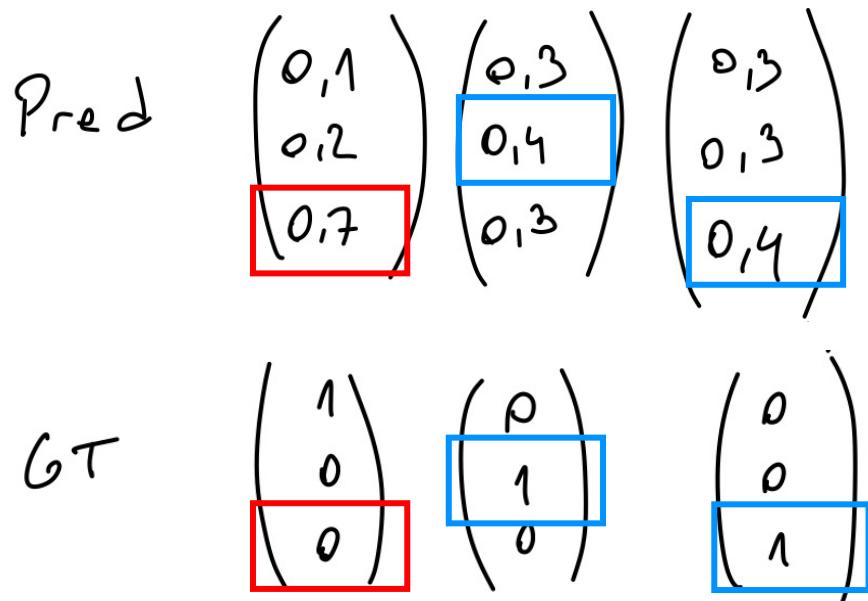
$$CE(p, q) = - \sum_x p(x) \log q(x)$$

Distribución Ground Truth

Distribución Estimada

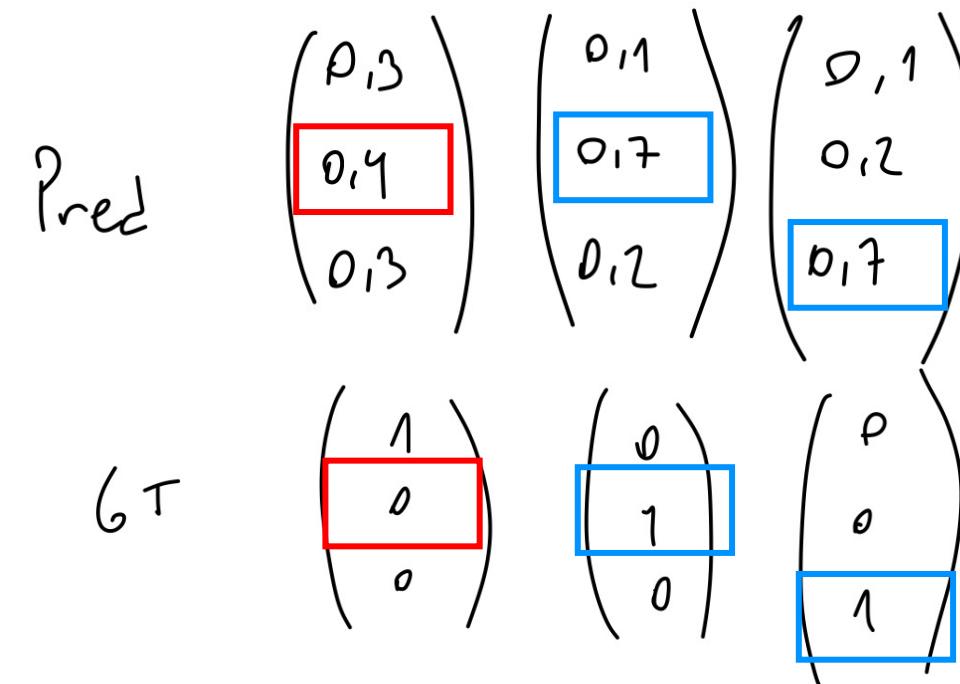
Función de pérdida

Entropía Cruzada vs Accuracy



Tasa de acierto = 2/3

Entropía cruzada = 4.14



Tasa de acierto = 2/3

Entropía cruzada = 1.92

Teorema de aproximación universal

Hornik et al, 1989

Neural Networks, Vol. 2, pp. 359–366, 1989
Printed in the USA. All rights reserved.

0893-6080/89 \$3.00 + .00
Copyright © 1989 Pergamon Press plc

ORIGINAL CONTRIBUTION

Multilayer Feedforward Networks are Universal Approximators

KURT HORNIK

Technische Universität Wien

MAXWELL STINCHCOMBE AND HALBERT WHITE
University of California, San Diego

(Received 16 September 1988; revised and accepted 9 March 1989)

Abstract—This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.

Keywords—Feedforward networks, Universal approximation, Mapping networks, Network representation capability, Stone-Weierstrass Theorem, Squashing functions, Sigma-Pi networks, Back-propagation networks.

any function encountered in applications leads one to wonder about the ultimate capabilities of such networks. In this paper, we present some data re-

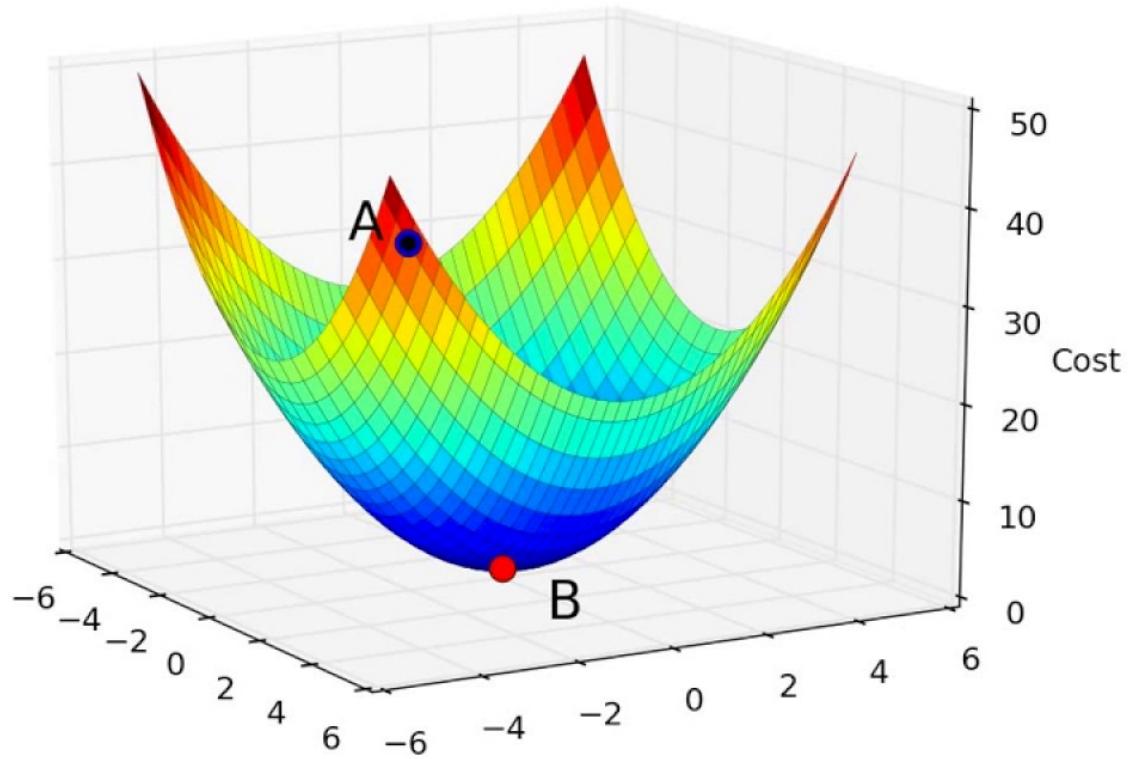
Teorema de aproximación universal

Hornik et al, 1989

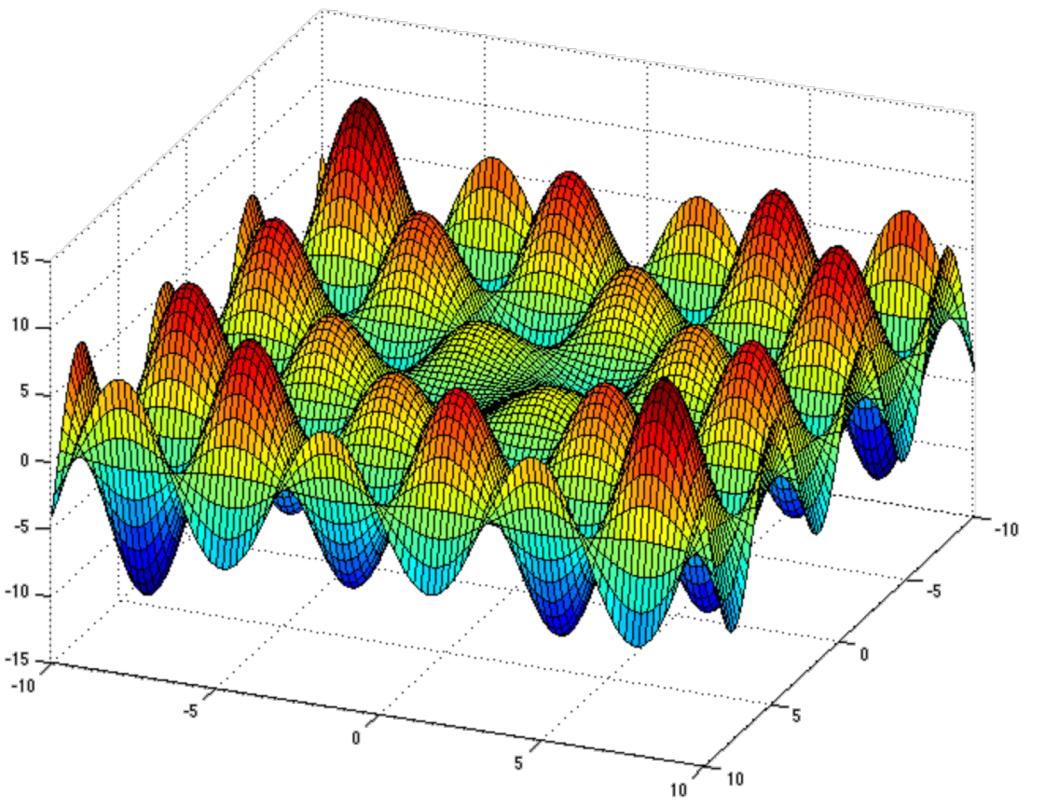
Establece que una **red feed-forward con una sola capa intermedia** es suficiente para **aproximar, con una precisión arbitraria, cualquier función** con un número finito de discontinuidades, **siempre y cuando las funciones de activación de las neuronas ocultas sean no lineales.**

Es un teorema de existencia (dice que la solución existe pero no dice que sea fácil encontrarla)

Función de pérdida no-convexa



Perceptrón simple, caso lineal
Convexa



Perceptrón multicapa con
funciones de activación no lineales
No convexa

Cómo entrenar mi perceptrón multicapa?

Función de pérdida no-convexa en MLP

- Gradient descent **no garantiza encontrar el mínimo global** en funciones no-convexas
- Sin embargo, **para redes neuronales grandes, la mayoría de los mínimos locales son similares** y presentan **performance similar en los dataset de test**
- La probabilidad de encontrar mínimos locales 'malos' decrece con el tamaño de la red
- **Focalizarse demasiado en la búsqueda del mínimo global en el dataset de entrenamiento** no es útil en la práctica, y puede terminar en **sobreajuste** del modelo

Cómo calcular el gradiente?

- **1. Derivación analítica:** derivar a mano y escribir el código
- **2. Derivación numérica :** diferencias finitas
- **3. Derivación simbólica:** se realiza utilizando las reglas estudiadas en Análisis matemático pero automatizadas (ej: Maple, Mathematica) Backpropagation
- **4. Derivación automática:**
 - Se definen las derivadas para las operaciones 'primitivas' (matemáticas y de control)
 - Se construye un grafo de operaciones y se deriva siguiendo la regla de la cadena.

Cálculo del gradiente en redes feed forward

Backpropagation (o algoritmo de retropropagación)

Learning representations by back-propagating errors

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

* Institute for Cognitive Science, C-015, University of California,
San Diego, La Jolla, California 92093, USA

† Department of Computer Science, Carnegie-Mellon University,
Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like 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 vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure¹.

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for the task it is given. This can be done by specifying by giving the

more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input-output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j , to unit j is a linear function of the outputs, y_i , of the units that are connected to j and of the weights, w_{ji} , on these connections

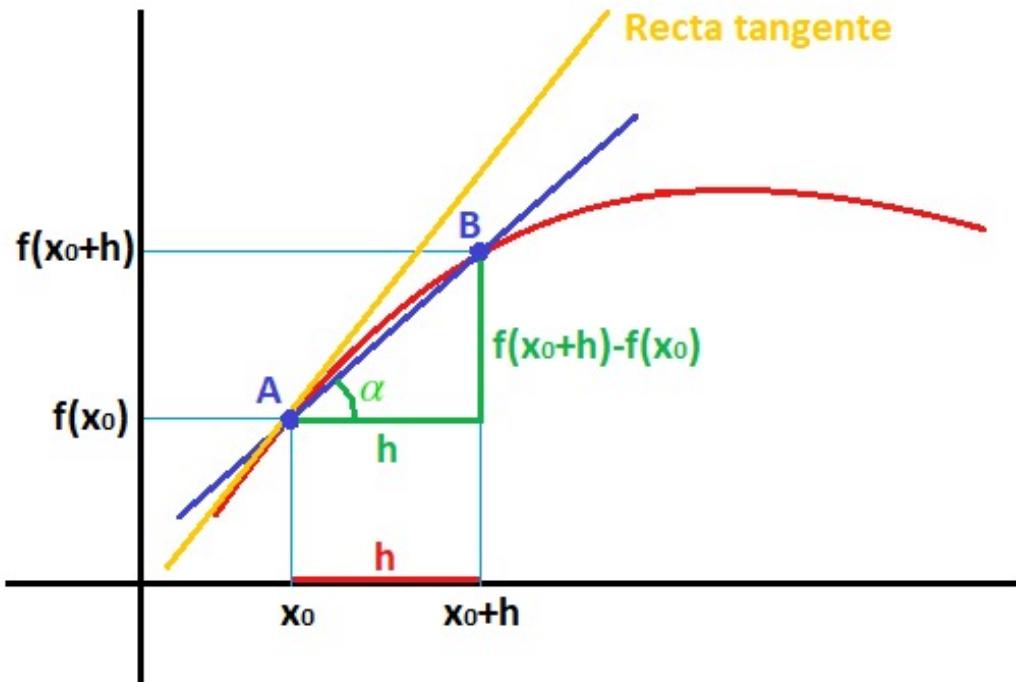
$$x_j = \sum_i y_i w_{ji} \quad (1)$$

These units can be given biases by introducing an extra input to each

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533.

Backpropagation

Empecemos por recordar la definición de derivada



$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

Cuando h es muuuuy pequeña, la función puede ser bien aproximada por una recta.
La pendiente de esa recta, es la derivada.

Intuición de derivada

La derivada nos indica entonces **la 'sensibilidad' de la función al cambio en la variable.** Esto se observa al reordenar la fórmula (considerando un h lo suficientemente pequeño):

$$f(x + h) = f(x) + h \frac{df(x)}{dx}$$

Ejemplo
producto:

$$f(x, y) = xy \quad \frac{df}{dy} = x \quad \frac{df}{dx} = y$$

Si tomamos $x=4$ e $y=-3$, qué pasa con f , alrededor de ese punto, al variar x ?

Al variar el valor de x una pequeña cantidad h , el valor de la función f se decrementará $3h$.

Intuición de derivada

La derivada nos indica entonces **la 'sensibilidad' de la función al cambio en la variable.**

Ejemplo suma:

$$f(x, y) = x + y \quad \frac{df}{dy} = 1 \quad \frac{df}{dx} = 1$$

Si tomamos $x=4$ e $y=-3$, qué pasa con f , alrededor de ese punto, al variar x e y ?

En ambos casos la función se incrementa de igual forma, independientemente del valor de x y de y (a diferencia del caso del producto).

Intuición de derivada

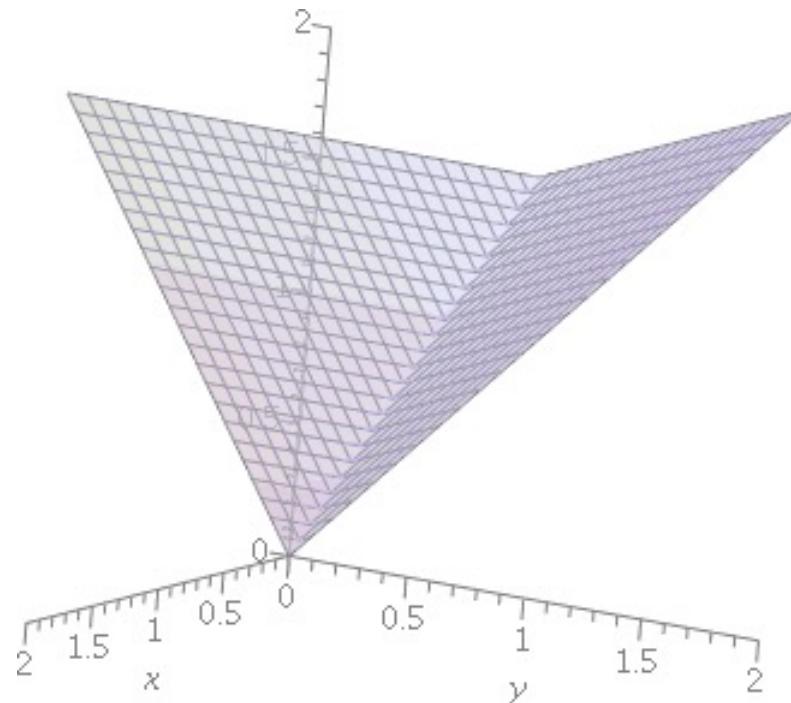
La derivada nos indica entonces **la 'sensibilidad' de la función al cambio en la variable.**

Ejemplo
Máximo:

$$f(x, y) = \max(x, y)$$

$$\frac{\partial f}{\partial x} = \mathbb{1}(x \geq y)$$

$$\frac{\partial f}{\partial y} = \mathbb{1}(y \geq x)$$



Función no diferenciable, por lo tanto hablamos de **sub-gradiente**

Generalización del gradiente para funciones no diferenciables

Cálculo del gradiente en redes feed forward

Backpropagation (o algoritmo de retropropagación)

Regla de la cadena

$$f = (x + y) * z$$

$$q = x + y$$

$$\begin{array}{ccc} & \longrightarrow & \\ q & \longrightarrow & f = q * z \end{array}$$

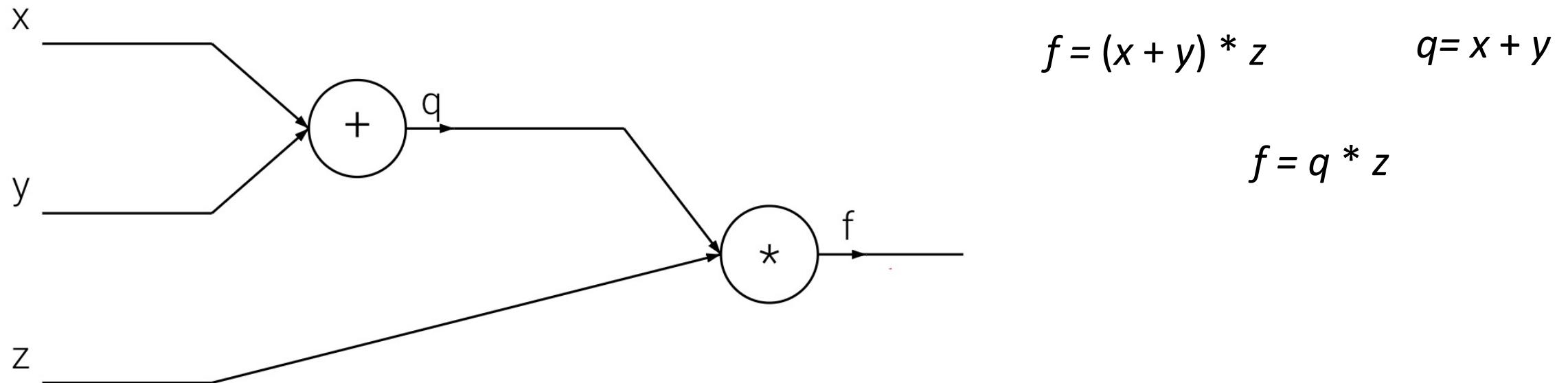
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Cálculo del gradiente en redes feed forward

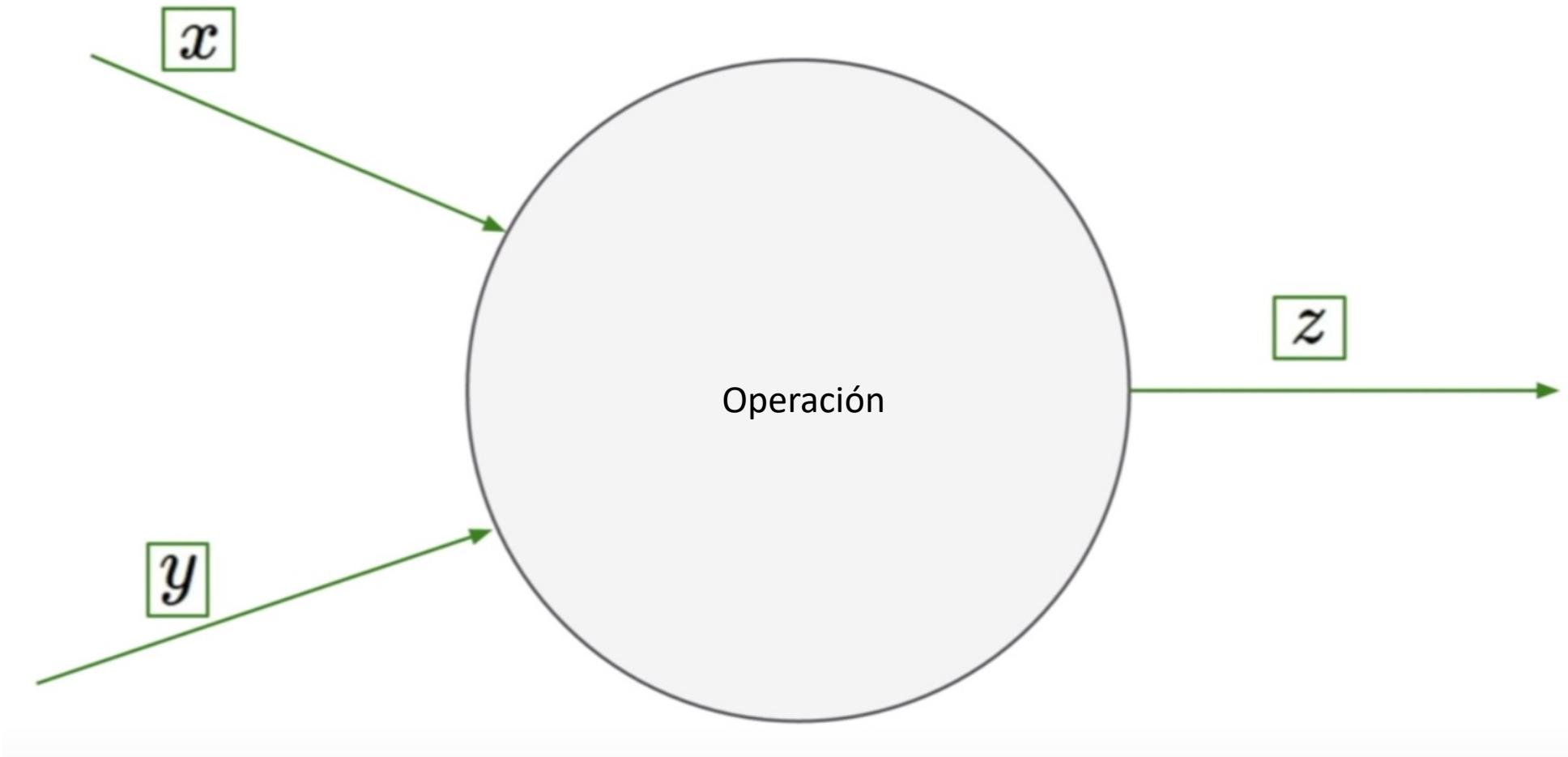
Backpropagation (o algoritmo de retropropagación)

Nuestro enfoque: Backpropagation como flujo de gradientes encadenados en un circuito de operaciones aritméticas

(Explicación basada en <http://cs231n.github.io>)

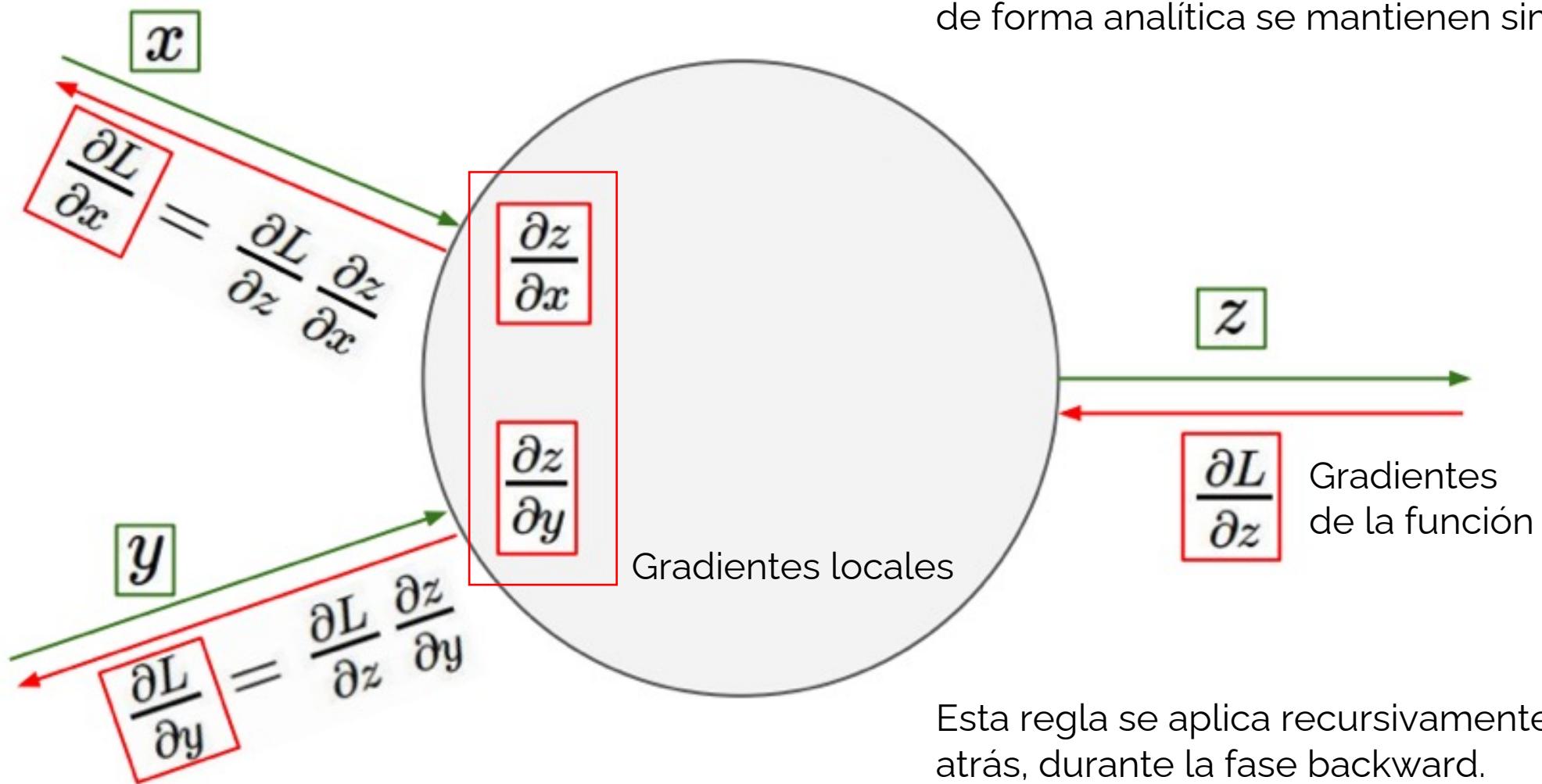


Flujo de gradientes



Flujo de gradientes

De esta forma, las operaciones a derivar de forma analítica se mantienen simples

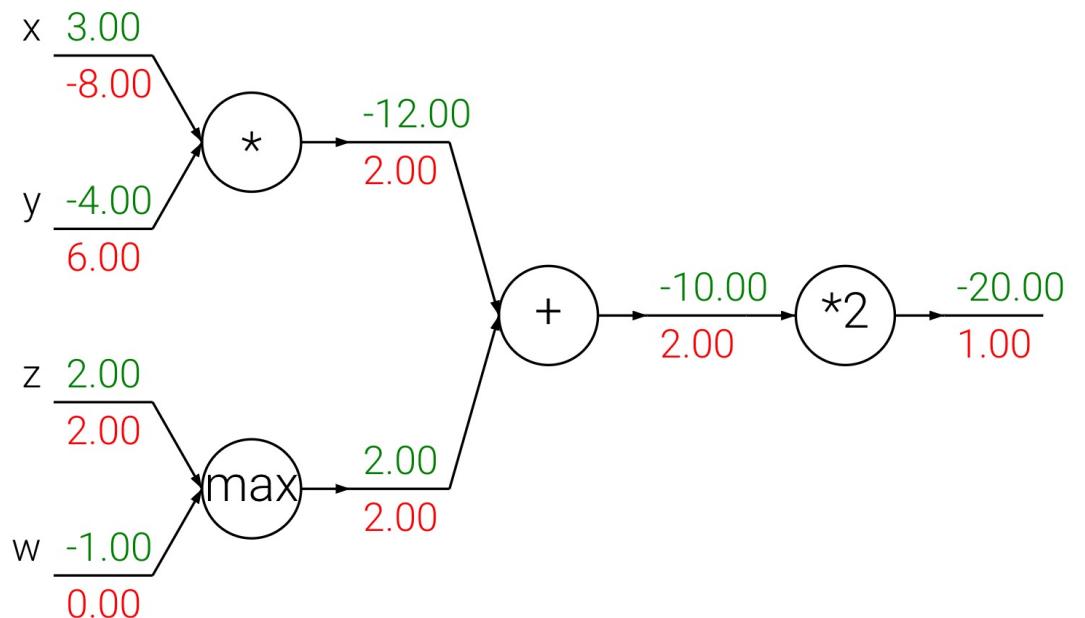


Ejemplos en pizarrón

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

$$f(x, y, z, w) = ((xy) + \max(z, w)) * 2$$

Patrones en el grafo de operaciones



- La **suma** distribuye el gradiente de igual forma a todas sus entradas.
- La operación **max** envía el gradiente sólo hacia la mayor entrada.
- La operación **producto** multiplica la otra entrada por el gradiente retropropagado.

Frameworks que implementan diferenciación automática



TensorFlow

P Y TORCH C H

Método del gradiente descendiente por mini-batches + backpropagation

1. Inicializar los pesos \mathbf{w} aleatoriamente

2. Dado el dataset $\mathcal{D} = \{(\mathbf{x}, d)_n\} = \{(\mathbf{x}, d)_1, \dots, (\mathbf{x}, d)_N\}$

3. Por cada minibatch $\mathcal{D}_M = \{(\mathbf{x}, d)_m\} = \{(\mathbf{x}, d)_1, \dots, (\mathbf{x}, d)_M\}, M < N$

Actualizar en
la dirección opuesta
al gradiente de la
pérdida promedio

$$\mathbf{W} = \mathbf{W} - \delta \boxed{\nabla_{\mathbf{W}} \mathcal{L}(\mathcal{D}_M; \mathbf{W})}$$

4. Volver a 2 hasta satisfacer algún criterio de convergencia

En la práctica se utiliza gradiente
descendiente por minibatches, y
el gradiente es calculado
utilizando backpropagation

Clase 2.B

Introducción a PyTorch

Enzo Ferrante

 eferrante@sinc.unl.edu.ar

 @enzoferante



Slides basadas en la presentación de Joaquín Seia

“PyTorch es un framework de deep learning open source desarrollado para ser flexible y modular para la investigación, pero con la estabilidad y el soporte necesarios para su uso en producción.” *

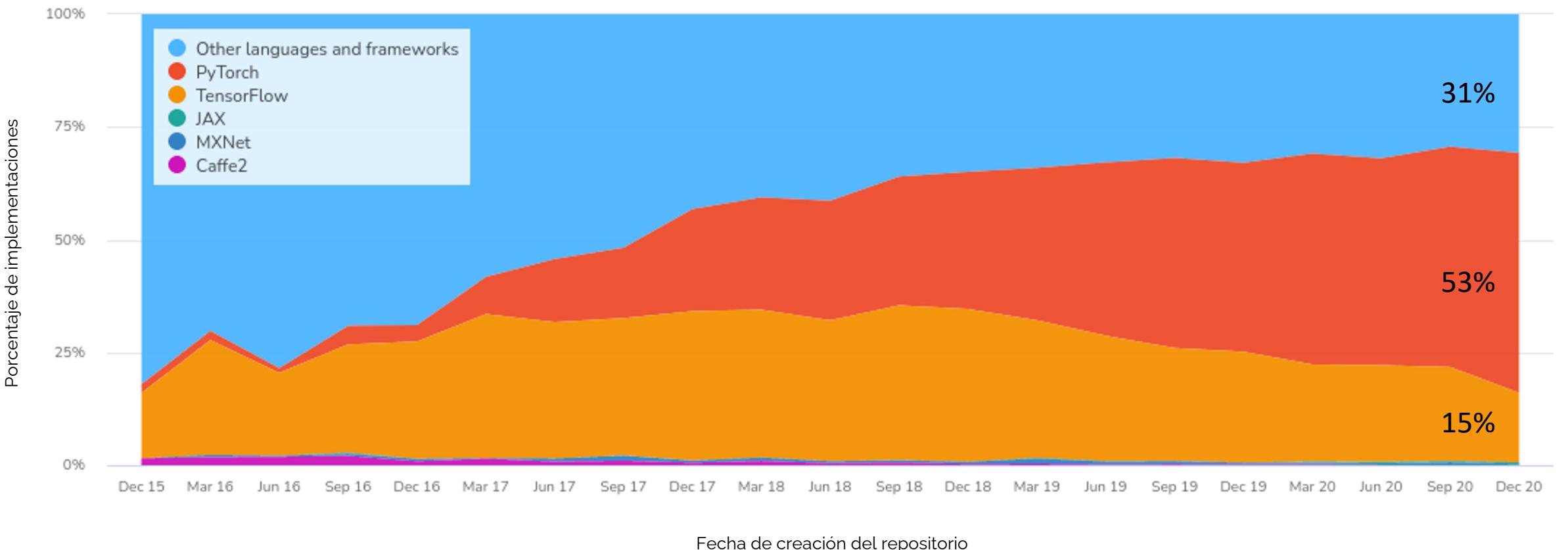


[facebook](#)

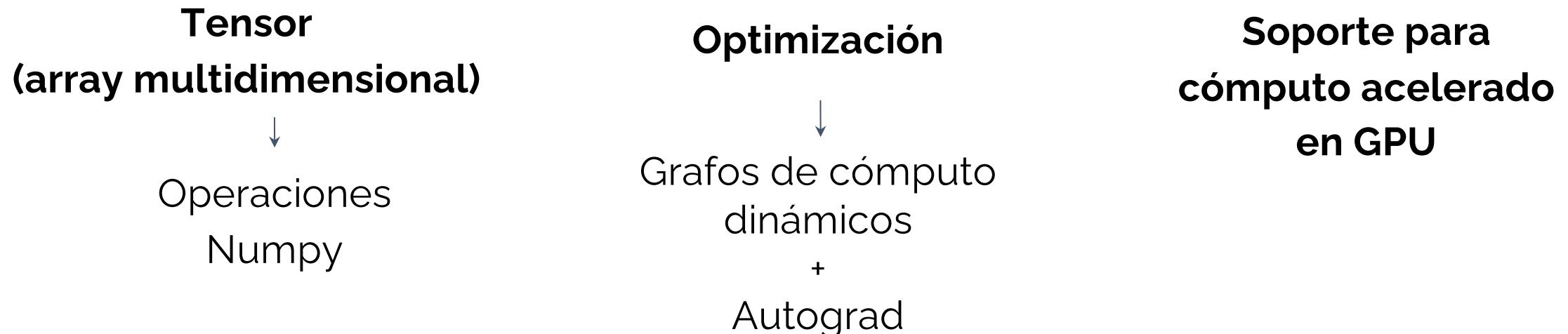
Artificial Intelligence Research

Adopción de PyTorch en los últimos años

Implementaciones de papers agrupadas por framework.*



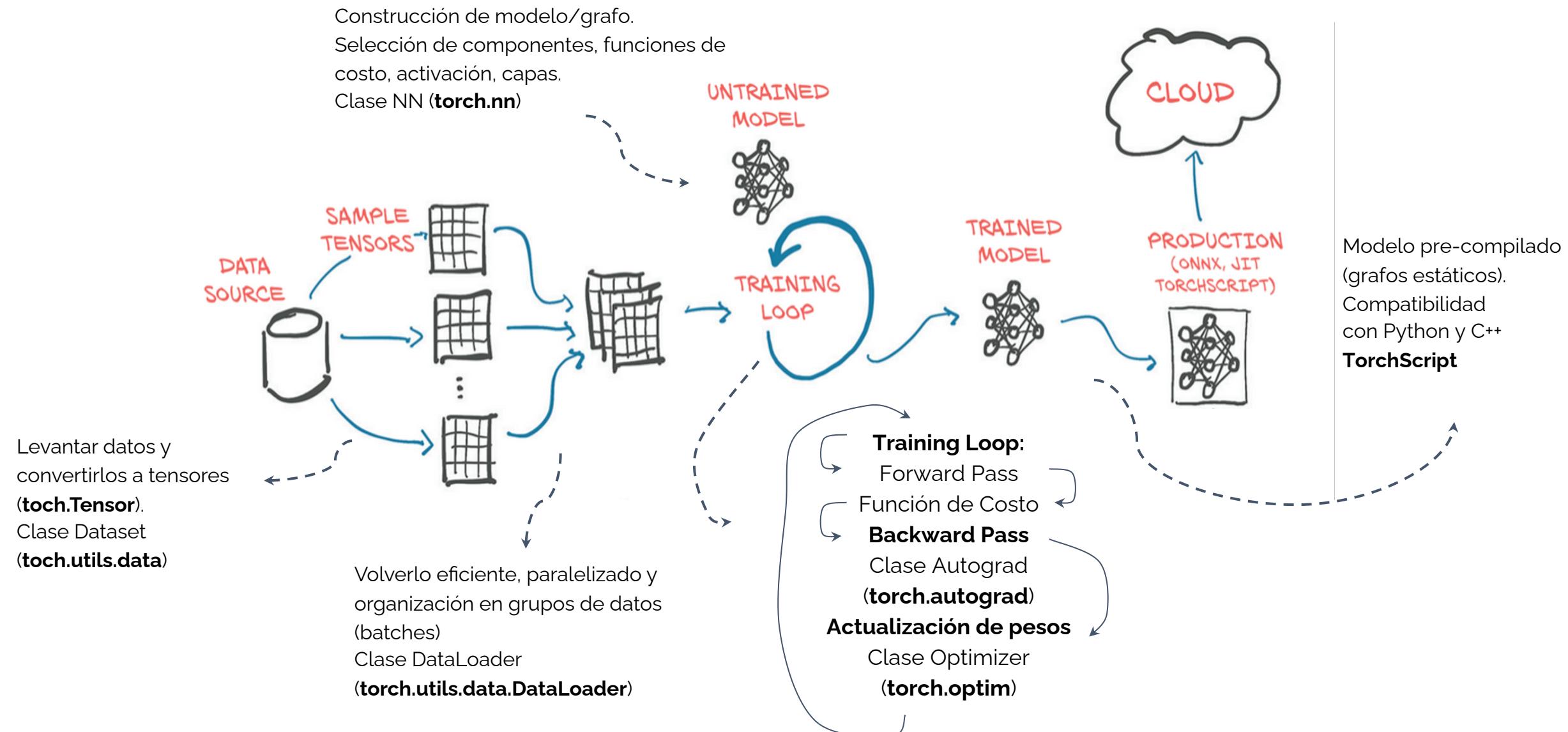
Fuentes: * <https://ai.facebook.com/tools/> ** <https://paperswithcode.com/trends>



Librería de alto desempeño que permite realizar optimización mediante **derivación automática** para desarrollos de cómputo científico en Python.

Deep Learning

PYTORCH



Tensores en PyTorch

```
import torch  
torch.tensor([[2, 3, 5], [1, 2, 9]])
```

```
tensor([[ 2,  3,  5],  
       [ 1,  2,  9]])
```

```
torch.rand(2, 2)
```

```
tensor([[ 0.0374, -0.0936],  
       [ 0.3135, -0.6961]])
```

```
a = torch.rand((3, 5))  
a.shape
```

```
torch.Size([3, 5])
```

```
import numpy as np  
np.array([[2, 3, 5], [1, 2, 9]])
```

```
array([[ 2,  3,  5],  
       [ 1,  2,  9]])
```

```
np.random.rand(2, 2)
```

```
array([[ 0.0374, -0.0936],  
       [ 0.3135, -0.6961]])
```

```
a = np.random.randn(3, 5)  
a.shape
```

```
(3, 5)
```

Tensores en PyTorch

Operaciones con Tensores

Creación
(ones, zeros)

Indexado,
slicing, joining

Matemáticas

Sampleo
Aleatorio
(randn, normal)

Serialización
(load, save)

Punto a punto
(abs, cos)

Reducción
(mean, std)

Comparación
(max, min)

Espectrales
(stft, etc)

Algebra
Lineal Basica
(BLAS)

Autograd

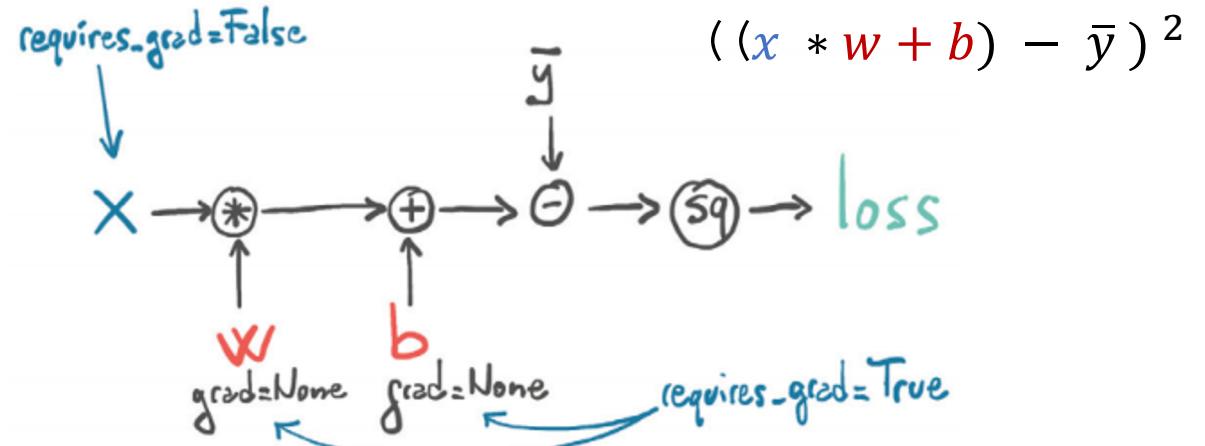
"Paquete de PyTorch que implementa diferenciación automática para funciones escalares arbitrarias por medio de la construcción de grafos computacionales dinámicos"

Tensores en PyTorch

- Computar operaciones en forma paralela en GPU
- Distribuir operaciones en múltiples máquinas
- Mantener un seguimiento del grafo de operaciones que les dan origen.

```
c = torch.rand(3).cuda()  
c = torch.rand(3).to('cuda')
```

```
c = torch.rand(3).cpu()  
c = torch.rand(3).to('cpu')
```



```
params = torch.tensor([1.0, 0.0], requires_grad=True)  
params  
tensor([1., 0.], requires_grad=True)
```

```
print(params.grad)
```

```
None
```

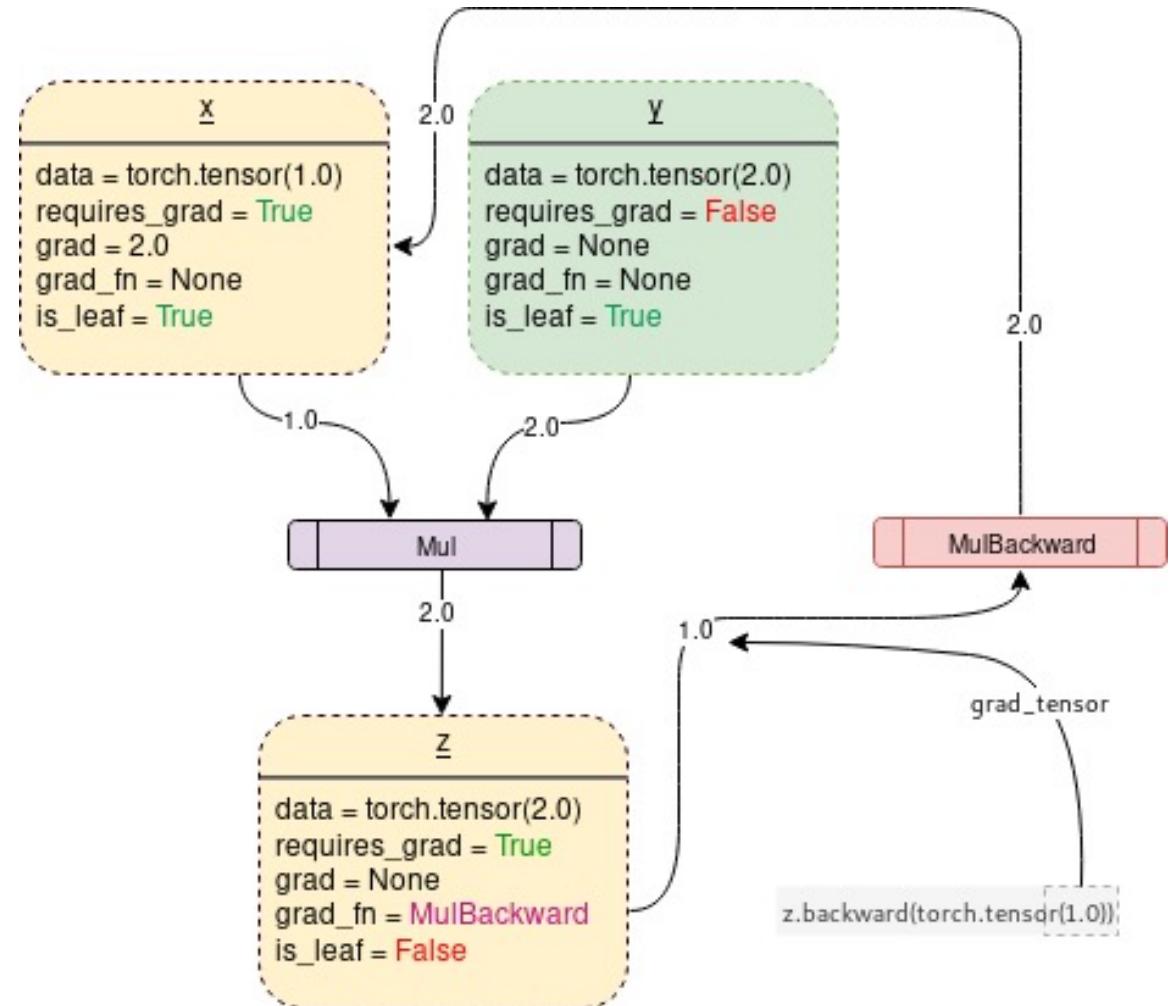
Dynamic Computational graph

```
import torch

# Pasada forward
x = torch.tensor([1.], requires_grad=True)
y = torch.tensor([2.], requires_grad=False)
z = x * y

print(x)
print(y)
print(z)

tensor([1.], requires_grad=True)
tensor([2.])
tensor([2.], grad_fn=<MulBackward0>)
```



Dynamic Computational graph

```
import torch

# Pasada forward
x = torch.tensor([1.], requires_grad=True)
y = torch.tensor([2.], requires_grad=False)
z = x * y

print(x)
print(y)
print(z)

tensor([1.], requires_grad=True)
tensor([2.])
tensor([2.], grad_fn=<MulBackward0>)
```

```
# Pasada backward
z.backward()
```

```
#Cálculo del gradiente
print("dz/dx: " + str(x.grad))
print("dz/dy: " + str(y.grad))
```

```
dz/dx: tensor([2.])
dz/dy: None
```

Autograd

```
# Definimos el modelo y la función de costo
def model(dato_ej, w, b):
    return w * dato_ej + b

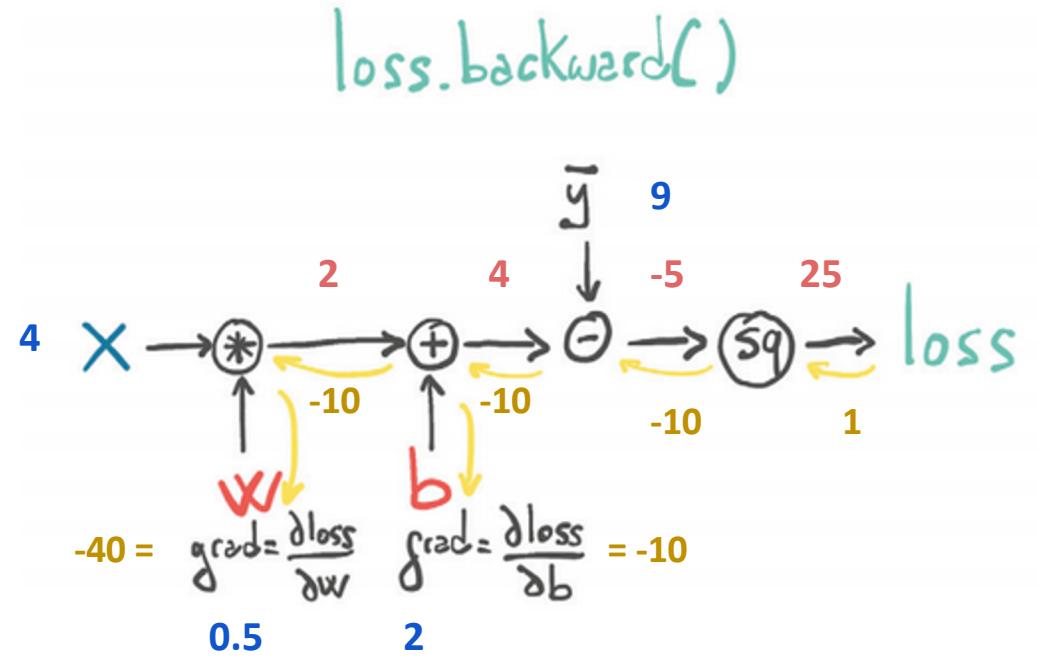
def loss_fn(pred, real):
    squared_diffs = (pred - real)**2
    return squared_diffs.mean()

# Definimos el ejemplo y los parametros
dato_ej = torch.tensor([4.], requires_grad=False)
w = torch.tensor([0.5], requires_grad=True)
b = torch.tensor([2.], requires_grad=True)
print(w.grad, b.grad)

# Forward pass
resultado_ej = model(dato_ej, w, b)
loss = loss_fn(resultado_ej, 9.)

# Backward pass
loss.backward()
print(w.grad, b.grad)

None None
tensor([-40.]) tensor([-10.])
```



OJO: ¡El método backward() acumula los gradientes en los nodos hoja!

Blucle de entrenamiento

```
# Ejemplo de un bucle de entrenamiento tradicional
for epoch in range(1, n_epochs + 1):
    # Se lleva a cero el gradiente acumulado
    if params.grad is not None:
        params.grad.zero_()

    # Forward pass
    prediccion = model(inputs, *params)
    # Función de costo
    loss = loss_fn(prediccion, ground_truth)
    # Backward pass
    loss.backward()

    # Con los gradientes, se actualizan los pesos
    with torch.no_grad():
        params -= learning_rate * params.grad

return params
```

```
# Ejemplo de un bucle de entrenamiento usando torch.optim
import torch.optim as optim

learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

for epoch in range(1, n_epochs + 1):
    # Se lleva a cero el gradiente acumulado
    optimizer.zero_grad() ←

    # Forward pass
    prediccion = model(inputs, *params)
    # Función de costo
    loss = loss_fn(prediccion, ground_truth)
    # Backward pass
    loss.backward()

    # Con los gradientes, se actualizan los pesos
    optimizer.step() ←
    return params
```

Optimizador

```
optimizer = torch.optim.SGD(lista_parametros, lr)
```

EJ: optimizer = torch.optim.SGD([x], lr=0.001)

Optimizando una función cuadrática

```
x = torch.tensor(.0, requires_grad=True)
optimizer = torch.optim.SGD([x], lr=0.001)
```

optimizer.zero_grad() ---> Setea en 0 los gradientes de los tensores

f = x ** 2 ---> Pasada forward que construye el grafo dinámico

f.backward() ---> Pasada backward que computa los gradientes

optimizer.step() ---> Avanza un "paso" y actualiza los parámetros con el correspondiente gradiente.

Modelos

```
import torch.nn as nn
import torch.nn.functional as F

# Network architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(5, 7, bias=False)
        self.fc2 = nn.Linear(7, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
net = Net()
print(net)
```

```
Net(
  (fc1): Linear(in_features=5, out_features=7, bias=False)
  (fc2): Linear(in_features=7, out_features=1, bias=True)
)
```

```
import torch.nn as nn
import torch.nn.functional as F

# Network architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.seq_model = nn.Sequential(
            nn.Linear(5, 7),
            nn.ReLU(),
            nn.Linear(7, 1)
        )
    def forward(self, x):
        x = self.seq_model(x)
        return x
net = Net()
print(net)
```

```
Net(
  (seq_model): Sequential(
    (0): Linear(in_features=5, out_features=7, bias=True)
    (1): ReLU()
    (2): Linear(in_features=7, out_features=1, bias=True)
  )
)
```

Modelos

```
# Ejemplo de un bucle de entrenamiento usando torch.optim
import torch.nn as nn
import torch.optim as optim

learning_rate = 1e-2
optimizer = optim.SGD(net.parameters(), lr=learning_rate)
loss_fn = nn.MSELoss()
for epoch in range(1, n_epochs + 1):
    # Se lleva a cero el gradiente acumulado
    optimizer.zero_grad()

    # Forward pass
    prediccion = net(inputs)
    # Función de costo
    loss = loss_fn(prediccion, ground_truth)
    # Backward pass
    loss.backward()

    # Con los gradientes, se actualizan los pesos
    optimizer.step()
```

Libro recomendado

