

Introducción al Aprendizaje Profundo



“Viendo” el progreso del aprendizaje profundo a lo largo de los años.



2015

Goodfeiów et al.



2018

Karras, laine, Aila



2020

Mit intro to Dwp Learning





Hi everybody, and welcome to MIT 6.S191





Hi everybody, and welcome to MIT 6.S191



1.1M views



3 months ago

That is easily the cleanest visual deepfake I've ever seen. It must have taken ages to render, because it just looks flawless.



2 months ago

WOW WOW WOW i am amazed.



5 months ago

THAT INTRO TO THE LECTURE IS SAVAGE!!!



3 months ago

This is the best example of a Course that sells itself. 🤖

2020

...crear este video de 2 minutos requería...

2 horas de audio profesional

50 horas de video en HD

Guion estático y predefinido

Más de \$1.5K USD en cómputo



2020

...crear este video de 2
minutos requería...

2 horas de audio profesional

50 horas de video en HD

Guion estático y predefinido

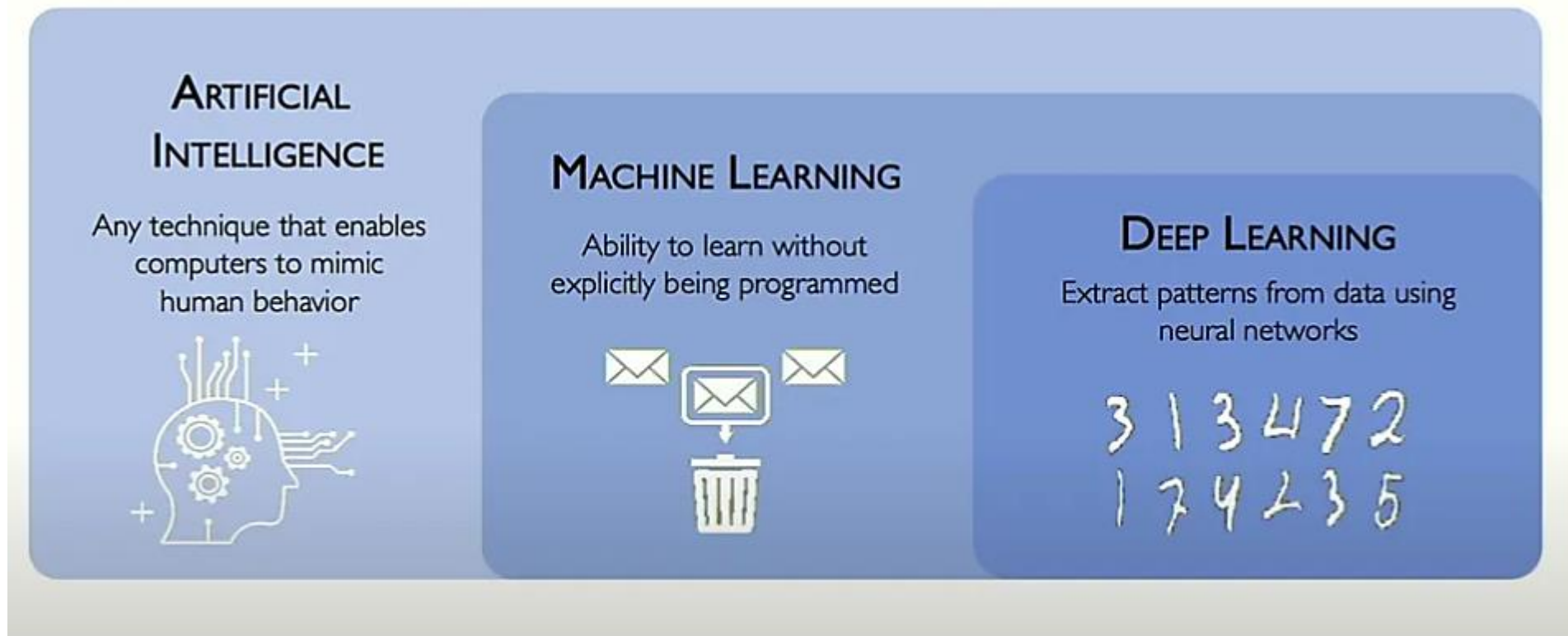
Más de \$1.5K USD en cómputo

2025

...avanzando unos años...



¿Qué es aprendizaje profundo?



Enseñar a las computadoras cómo aprender una tarea directamente a partir de datos en bruto.



Laboratorios de Software Actualizados



¿Por qué Aprendizaje
Profundo y por qué ahora?

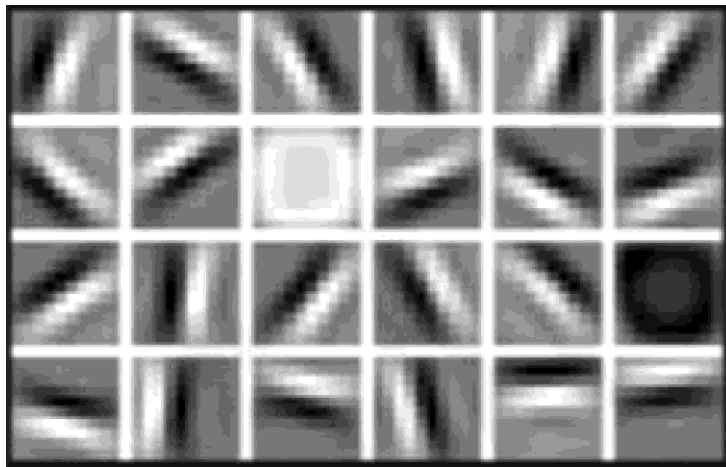


¿Por qué Aprendizaje Profundo?

Las características diseñadas manualmente consumen mucho tiempo, son frágiles y no escalables en la práctica.

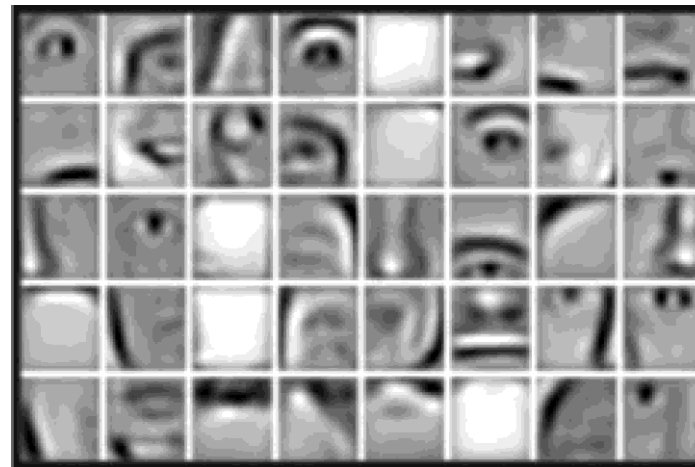
¿Podemos aprender las **características subyacentes** directamente desde los datos?

Características de bajo nivel



Líneas y bordes

Características de nivel medio



Ojos, nariz y orejas

Características de Alto Nivel

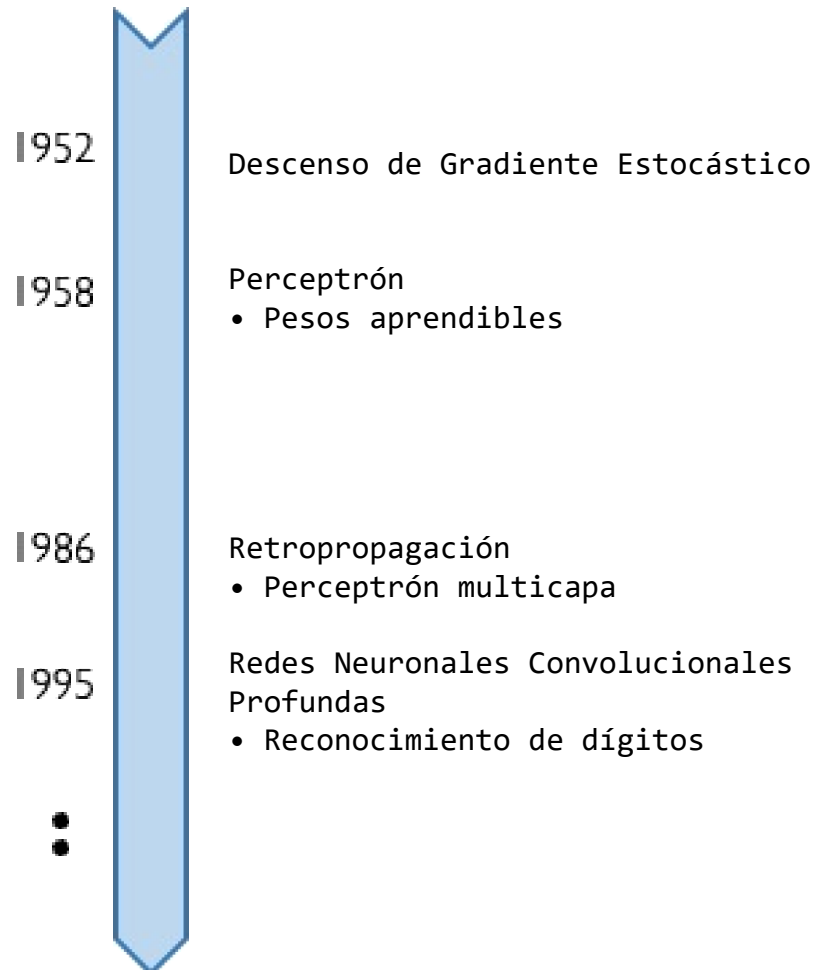


Estructura facial



¿Por qué no?

Las redes neuronales existen desde hace décadas, entonces
¿por qué su dominio actual?



1. Big Data

- Conjuntos de datos más grandes
- Recopilación y almacenamiento más fáciles



2. Hardware

- Unidades de procesamiento gráfico (GPUs)
- Altamente paralelizables



3. Software

- Técnicas mejoradas
- Nuevos modelos
- Herramientas y bibliotecas (Toolboxes)

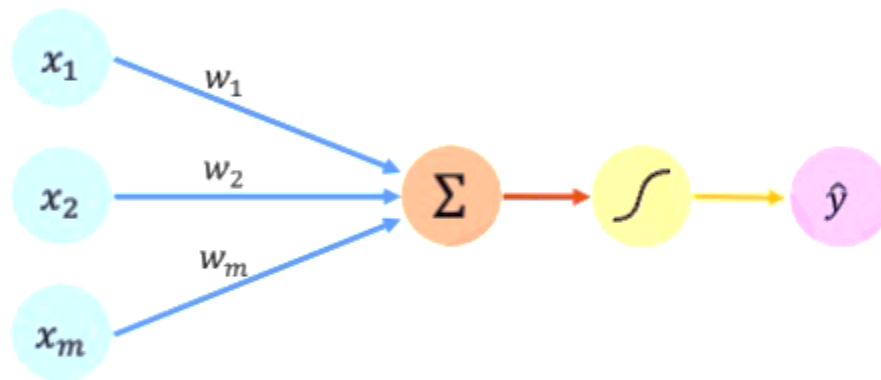


El Perceptrón

El bloque estructural fundamental del
aprendizaje profundo



El Perceptrón: Propagación Hacia Adelante



Inputs Weights Sum Non-Linearity Output

Output

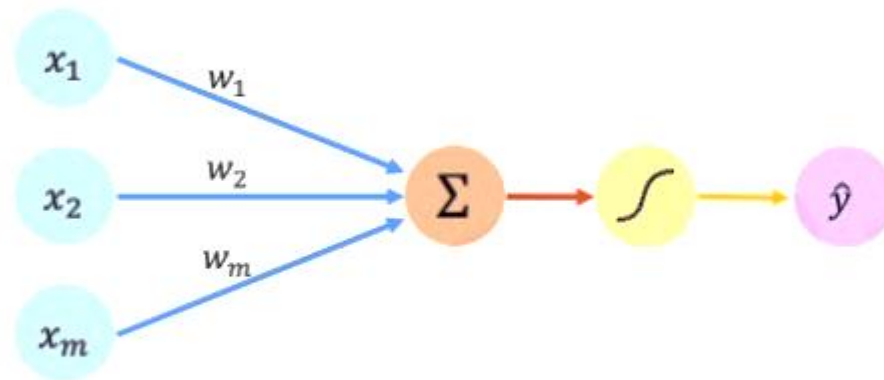
Linear combination of inputs

$$\hat{y} = g \left(\sum_{i=1}^m x_i w_i \right)$$

Non-linear activation function



El Perceptrón: Propagación Hacia Adelante



Inputs Weights Sum Non-Linearity Output

Output

Linear combination of inputs

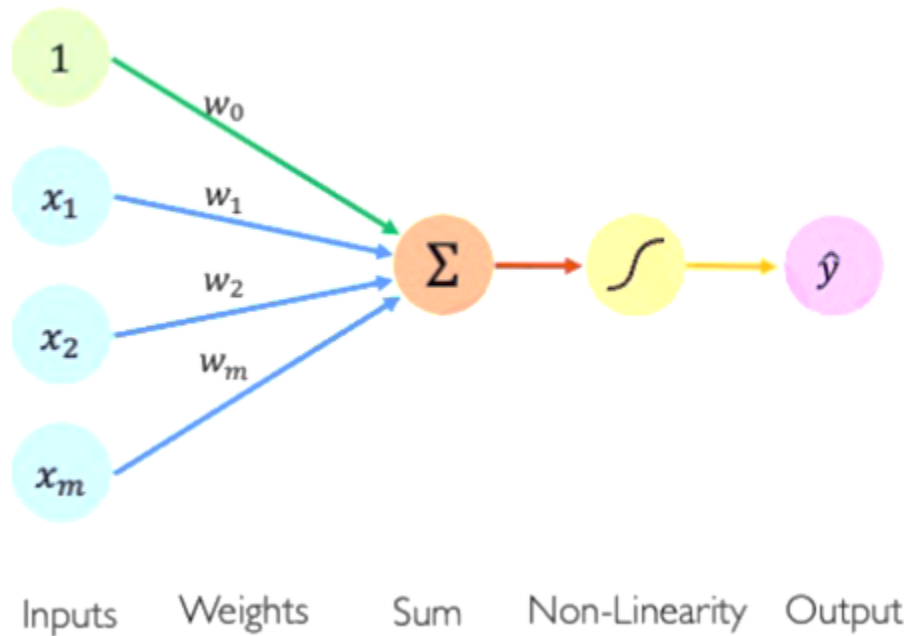
$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

Non-linear activation function

Bias



El Perceptrón: Propagación Hacia Adelante



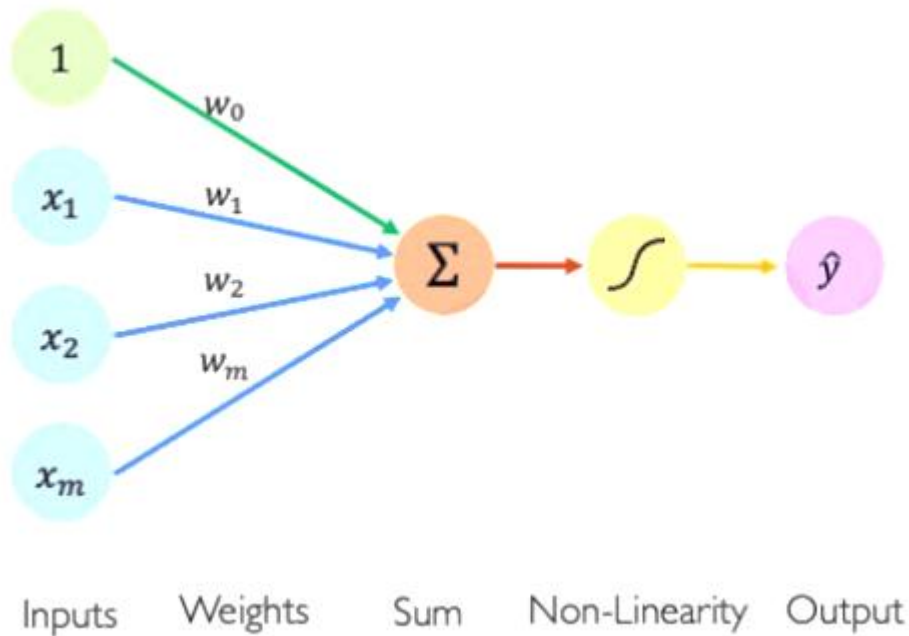
$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

$$\hat{y} = g (w_0 + \mathbf{X}^T \mathbf{W})$$

$$\text{where: } \mathbf{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } \mathbf{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$



El Perceptrón: Propagación Hacia Adelante

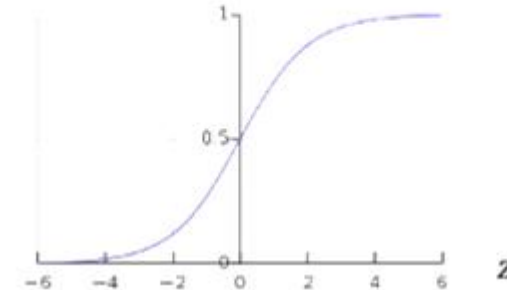


Activation Functions

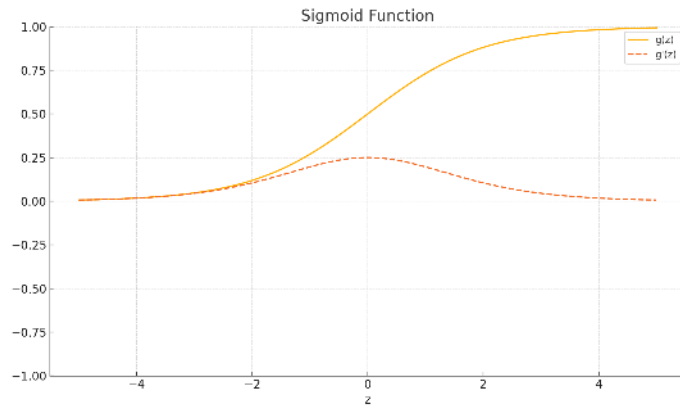
$$\hat{y} = g(w_0 + \mathbf{X}^T \mathbf{W})$$

- Example: sigmoid function

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Funciones de Activación Comunes



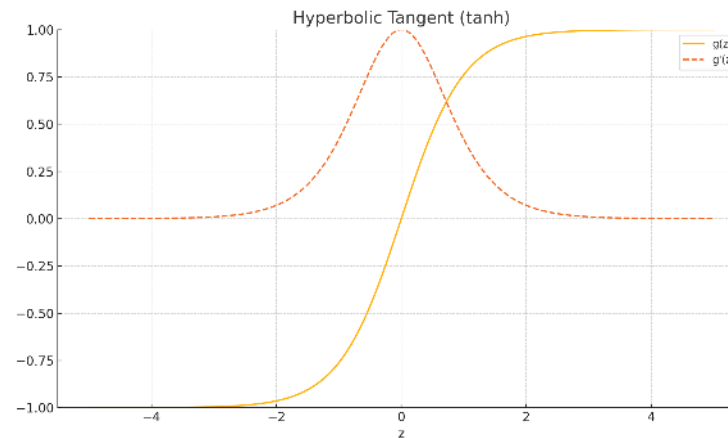
$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

```
tf.math.sigmoid(z)
```

```
torch.sigmoid(z)
```

 TensorFlow code blocks



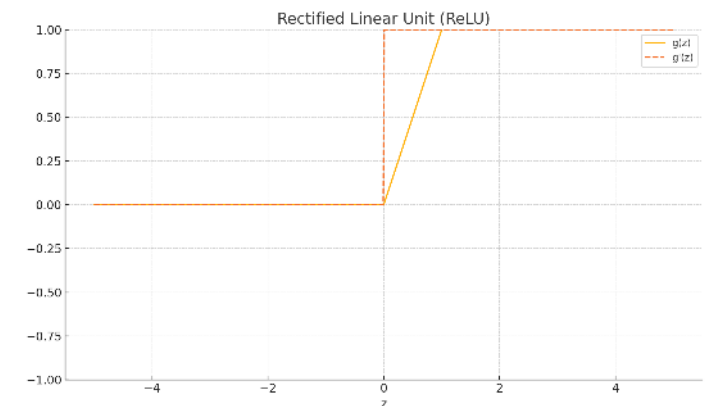
$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

```
tf.math.tanh(z)
```

```
torch.tanh(z)
```

NOTE: All activation functions are non-linear



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

```
tf.nn.relu(z)
```

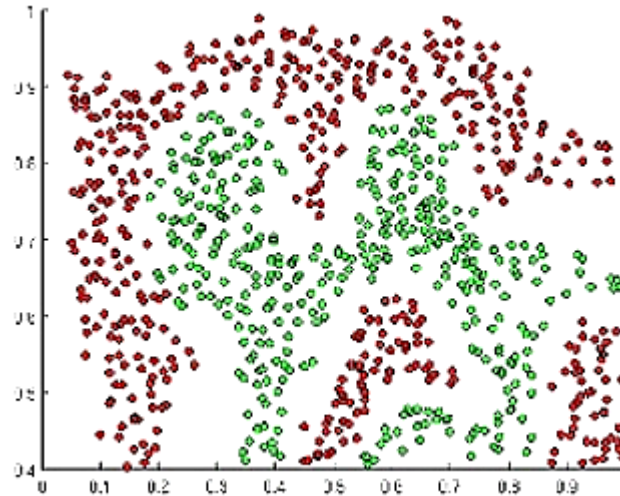
```
torch.nn.ReLU(z)
```

PyTorch code blocks 



Importancia de las Funciones de Activación

El propósito de las funciones de activación es introducir no linealidades en la red

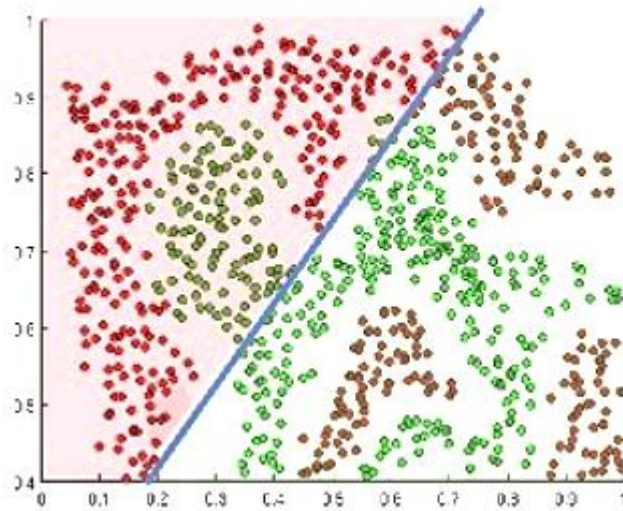


¿Qué pasaría si quisiéramos construir una red neuronal para distinguir los puntos verdes de los rojos?



Importancia de las Funciones de Activación

El propósito de las funciones de activación es introducir no linealidades en la red

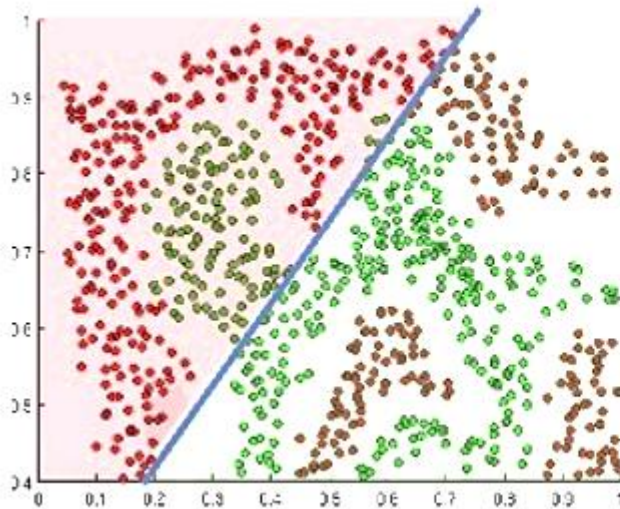


Las funciones de activación lineales producen decisiones lineales sin importar el tamaño de la red.

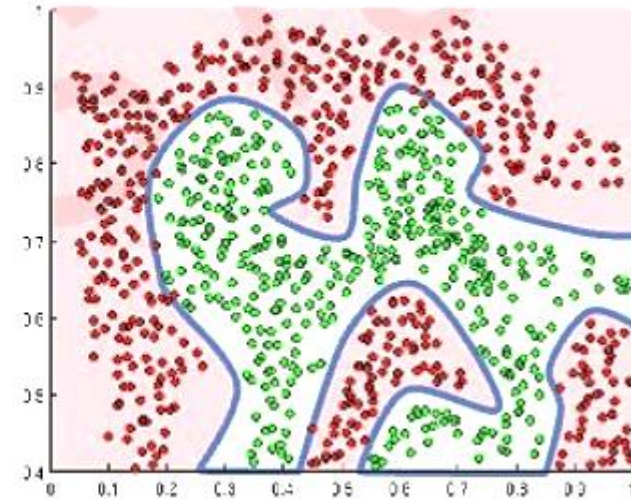


Importancia de las Funciones de Activación

El propósito de las funciones de activación es introducir no linealidades en la red



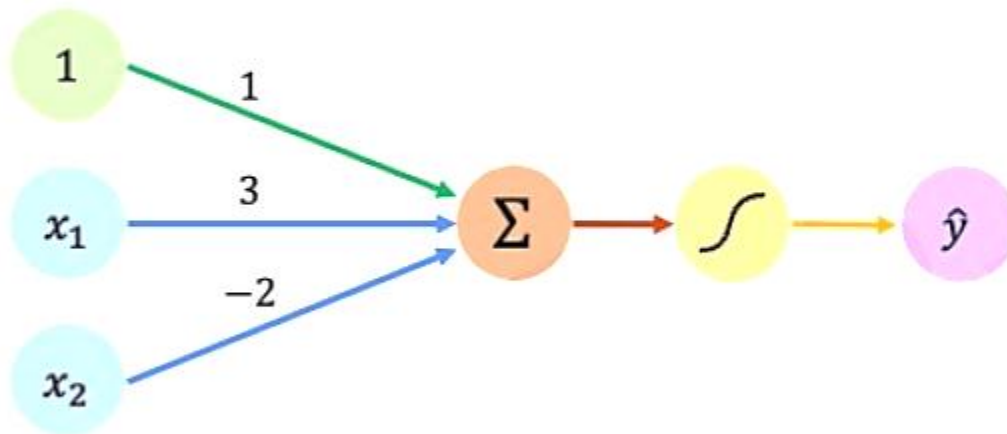
Las funciones de activación lineales producen decisiones lineales sin importar el tamaño de la red.



Las no linealidades nos permiten aproximar funciones arbitrariamente complejas.



El Perceptron: Ejemplo



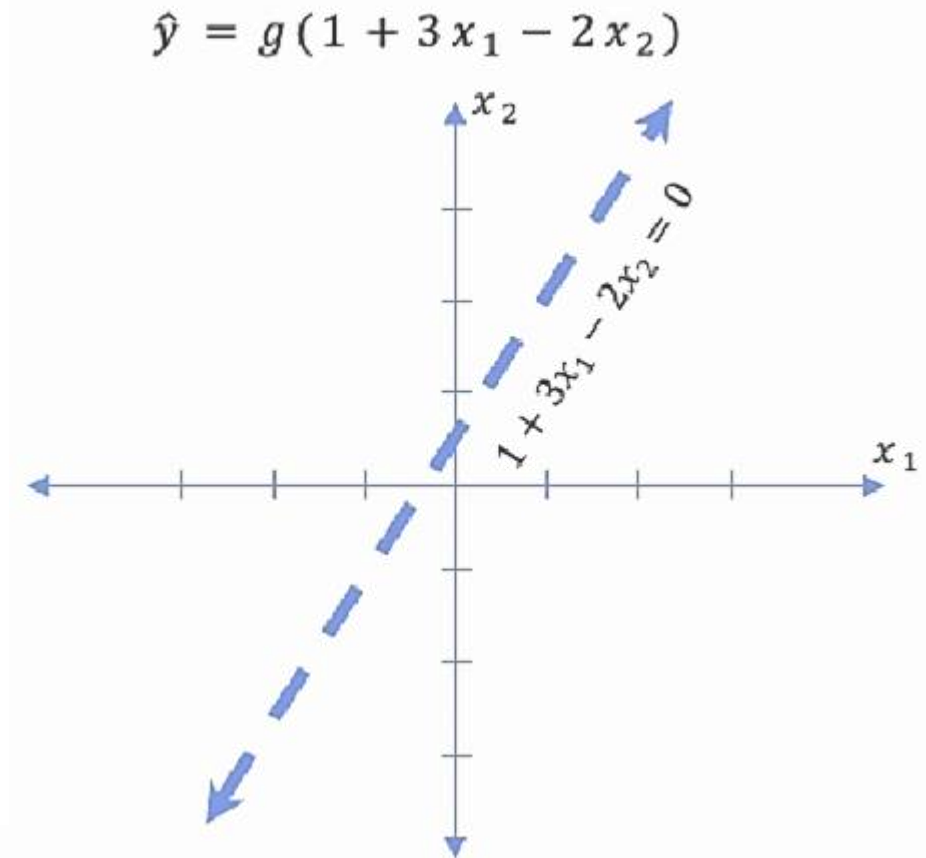
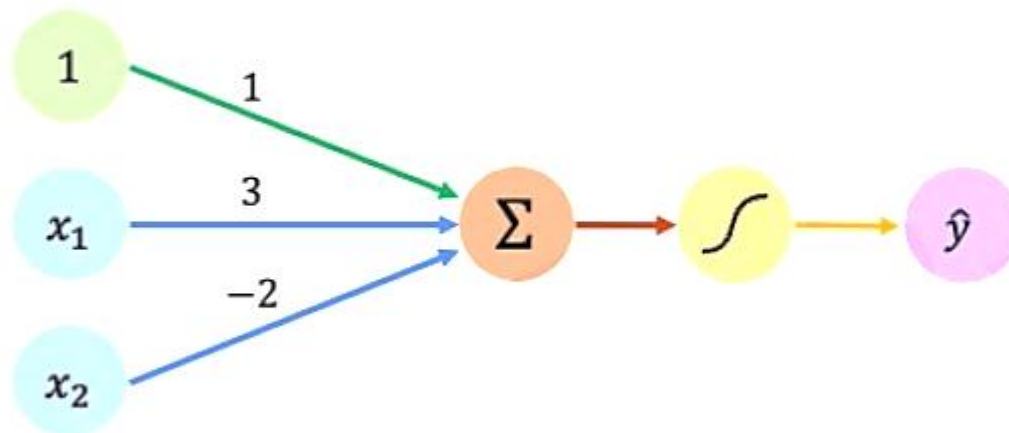
We have: $w_0 = 1$ and $\mathbf{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\begin{aligned}\hat{y} &= g(w_0 + \mathbf{X}^T \mathbf{W}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + 3x_1 - 2x_2)\end{aligned}$$

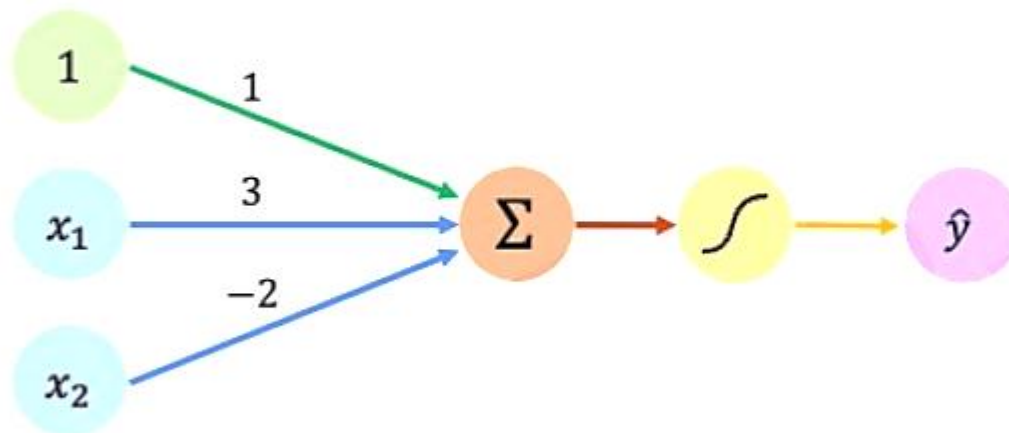
This is just a line in 2D!



El Perceptron: Ejemplo

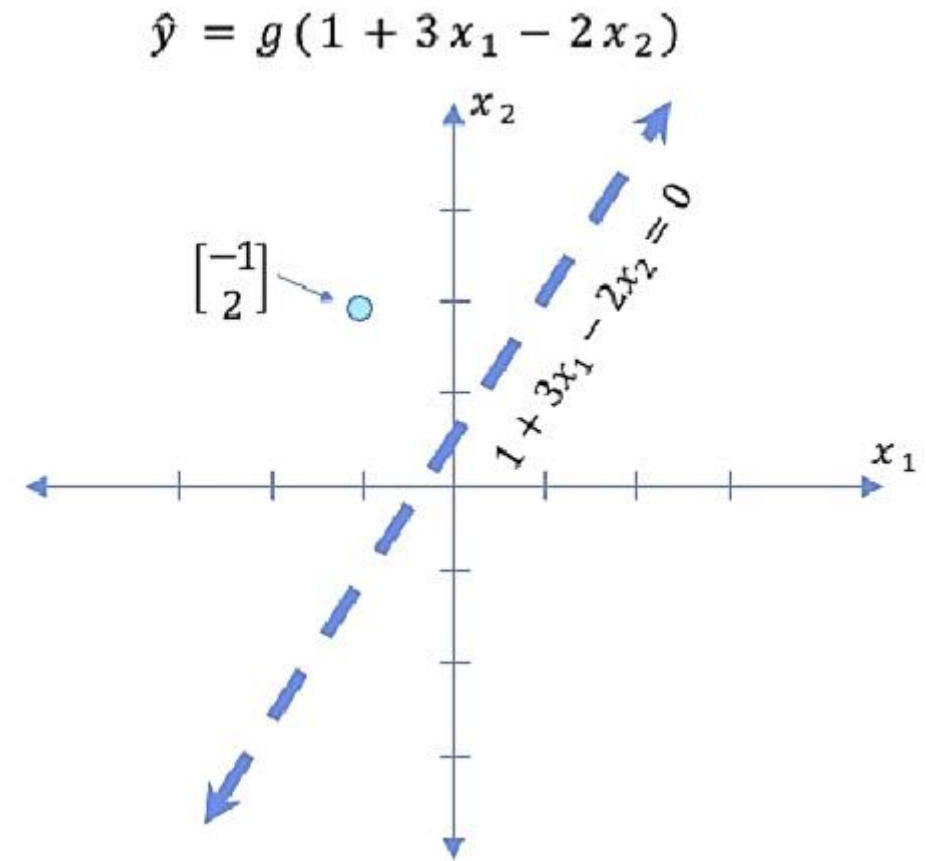


El Perceptron: Ejemplo

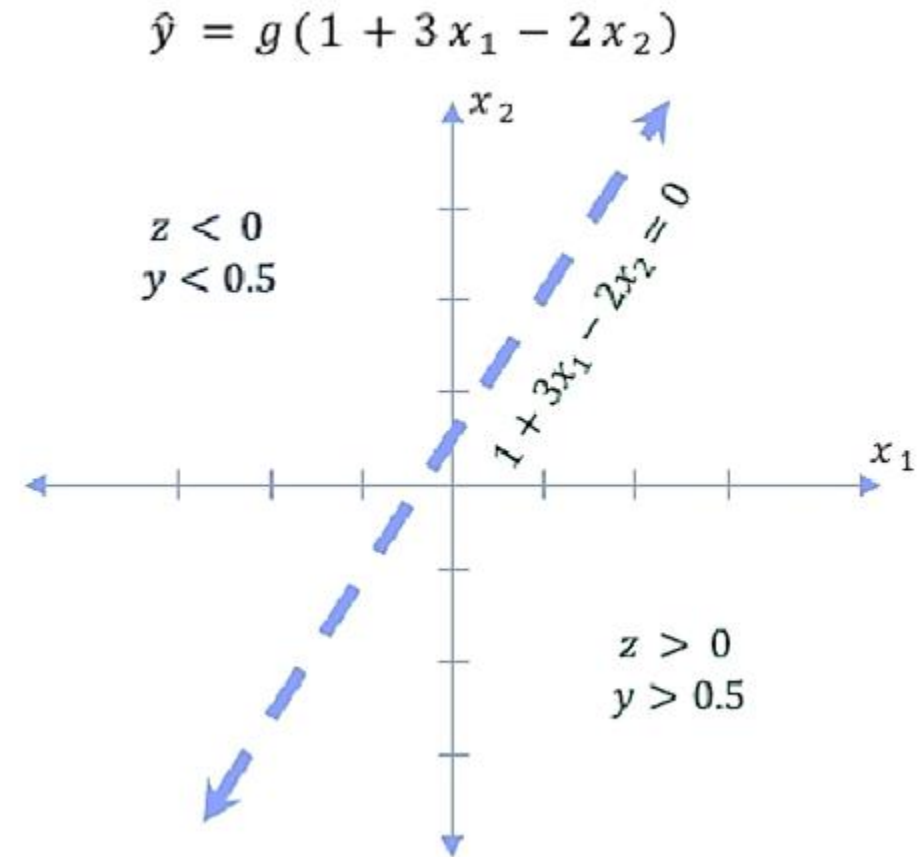
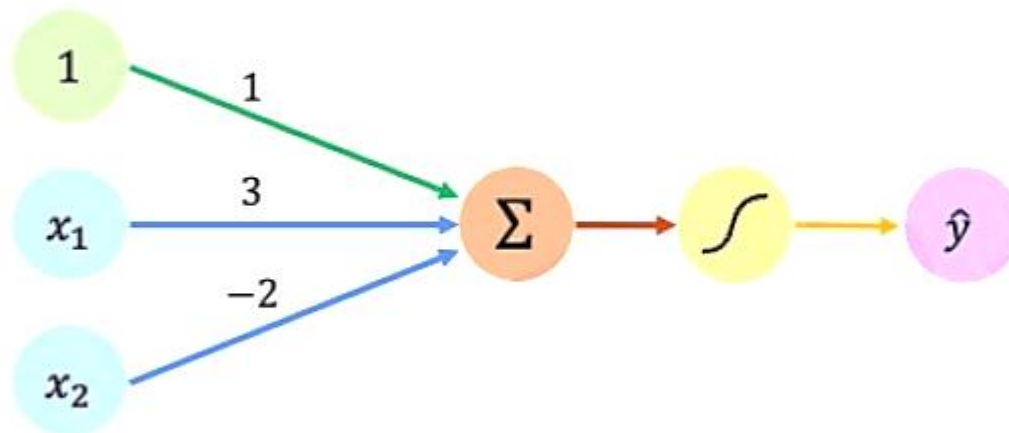


Assume we have input: $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\begin{aligned}\hat{y} &= g(1 + (3 * -1) - (2 * 2)) \\ &= g(-6) \approx 0.002\end{aligned}$$



El Perceptron: Ejemplo

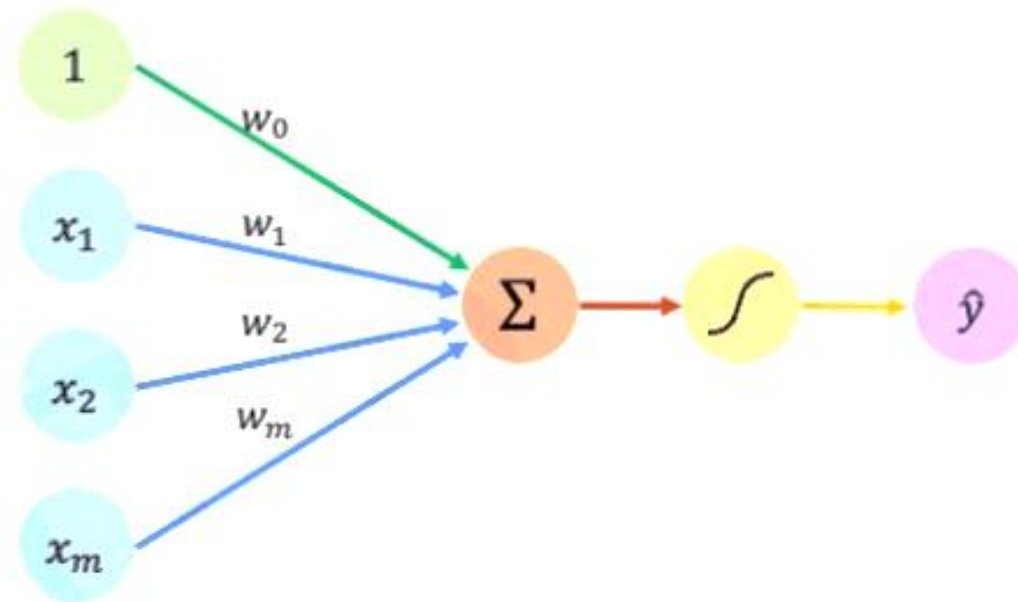


Construyendo Redes Neuronales con Perceptrones



El Perceptrón: Simplificado

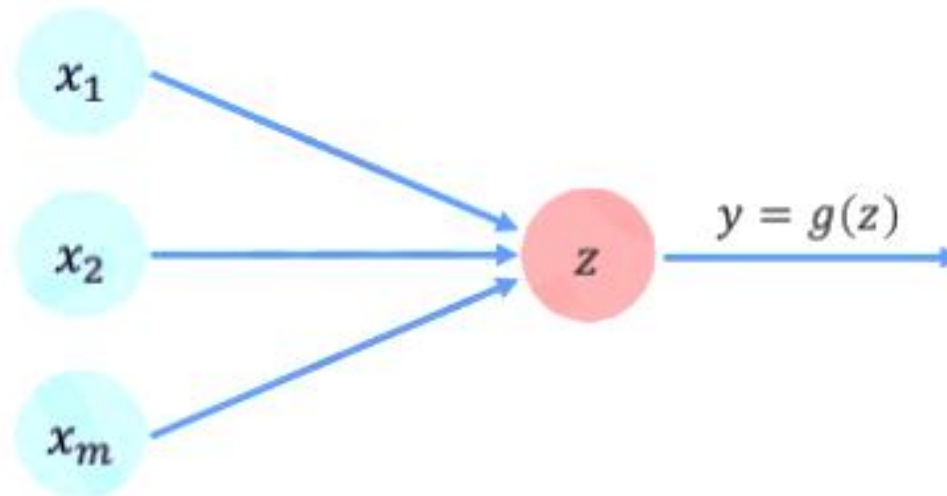
$$\hat{y} = g(w_0 + \mathbf{X}^T \mathbf{W})$$



Inputs Weights Sum Non-Linearity Output



El Perceptrón: Simplificado

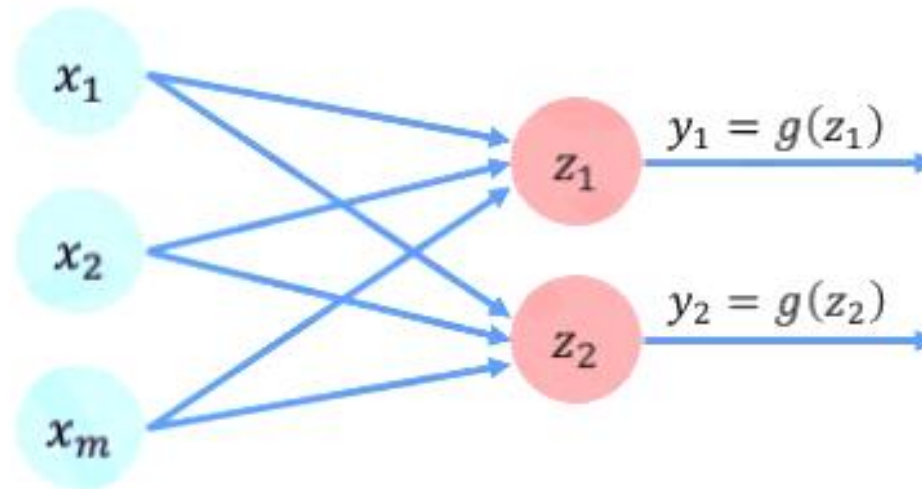


$$z = w_0 + \sum_{j=1}^m x_j w_j$$



Perceptrón de Salida Múltiple

Debido a que todas las entradas están densamente conectadas con todas las salidas, estas capas se denominan capas **densas**.



$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$



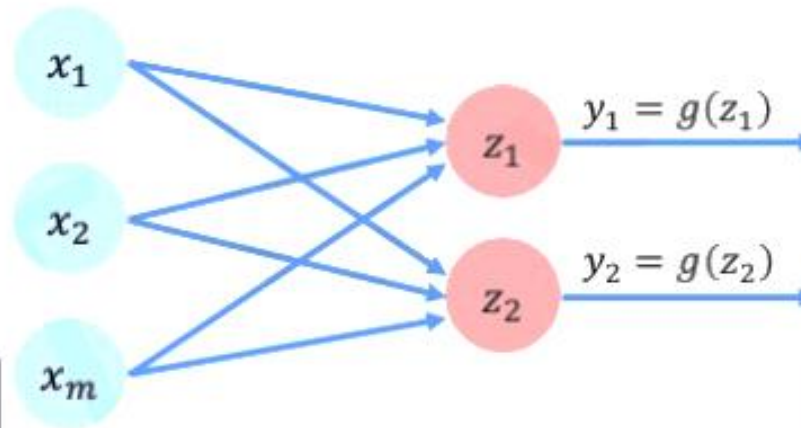
```
class MyDenseLayer(tf.keras.layers.Layer):  
    def __init__(self, input_dim, output_dim):  
        super(MyDenseLayer, self).__init__()  
  
        # Initialize weights and bias  
        self.W = self.add_weight([input_dim, output_dim])  
        self.b = self.add_weight([1, output_dim])  
  
    def call(self, inputs):  
        # Forward propagate the inputs  
        z = tf.matmul(inputs, self.W) + self.b  
  
        # Feed through a non-linear activation  
        output = tf.math.sigmoid(z)  
  
        return output
```

```
import tensorflow as tf  
  
layer = tf.keras.layers.Dense(units=2)
```



Perceptrón de Salida Múltiple

Debido a que todas las entradas están densamente conectadas con todas las salidas, estas capas se denominan capas densas



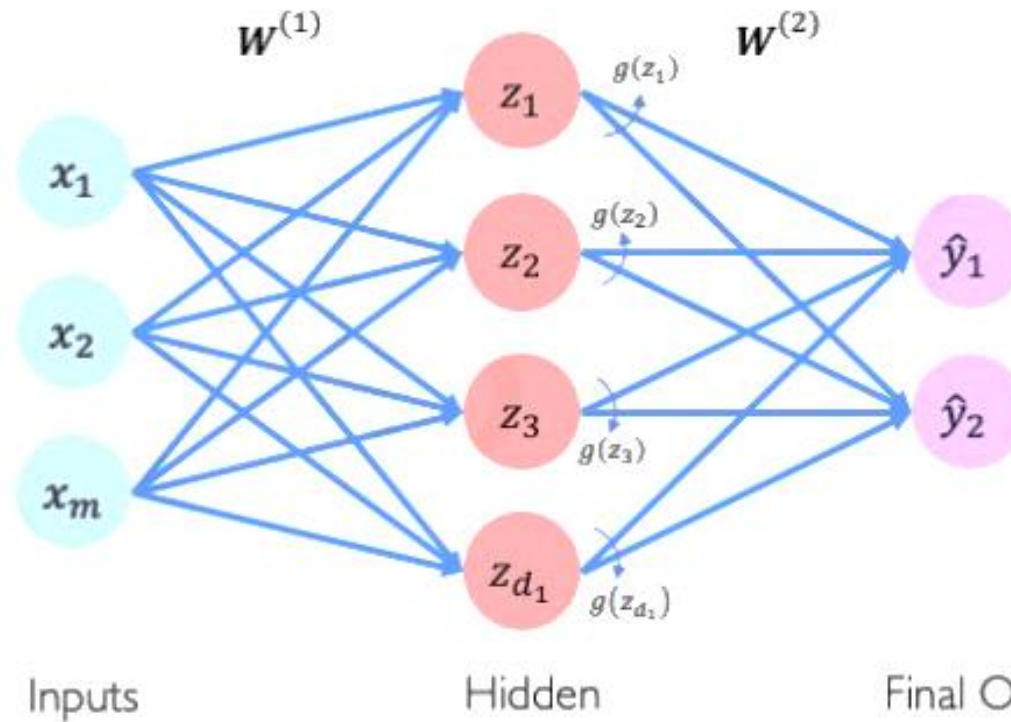
```
import tensorflow as tf  
  
layer = tf.keras.layers.Dense(  
    units=2)
```

```
import torch.nn as nn  
  
layer = nn.Linear(in_features=m,  
    out_features=2)
```

$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$



Red Neuronal de Una Sola Capa

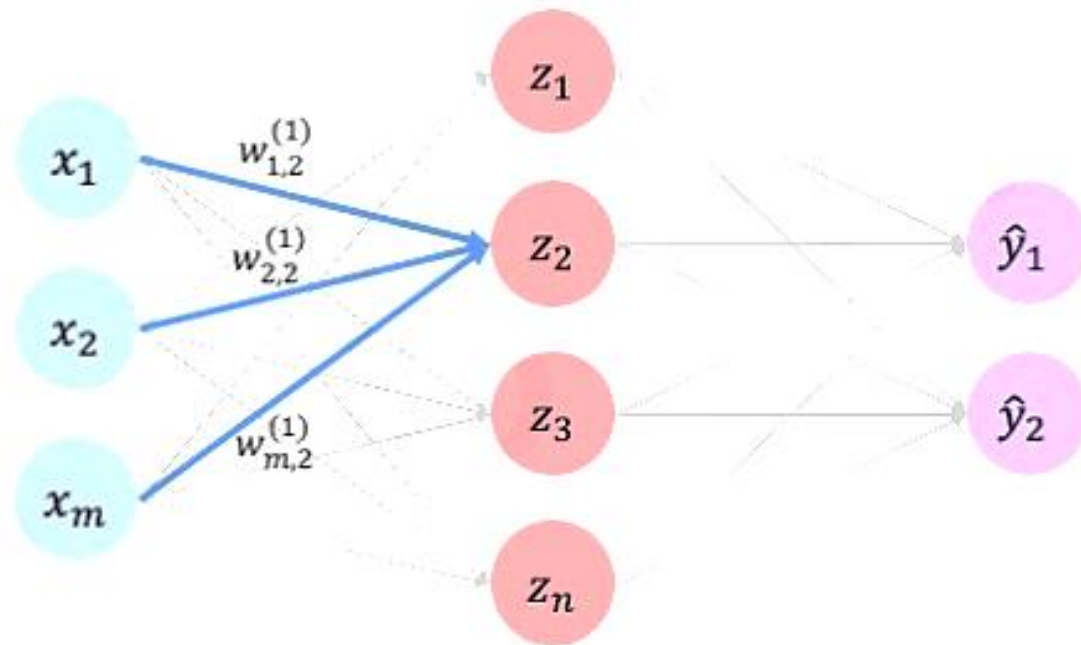


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^m x_j w_{j,i}^{(1)}$$

$$\hat{y}_i = g \left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) w_{j,i}^{(2)} \right)$$



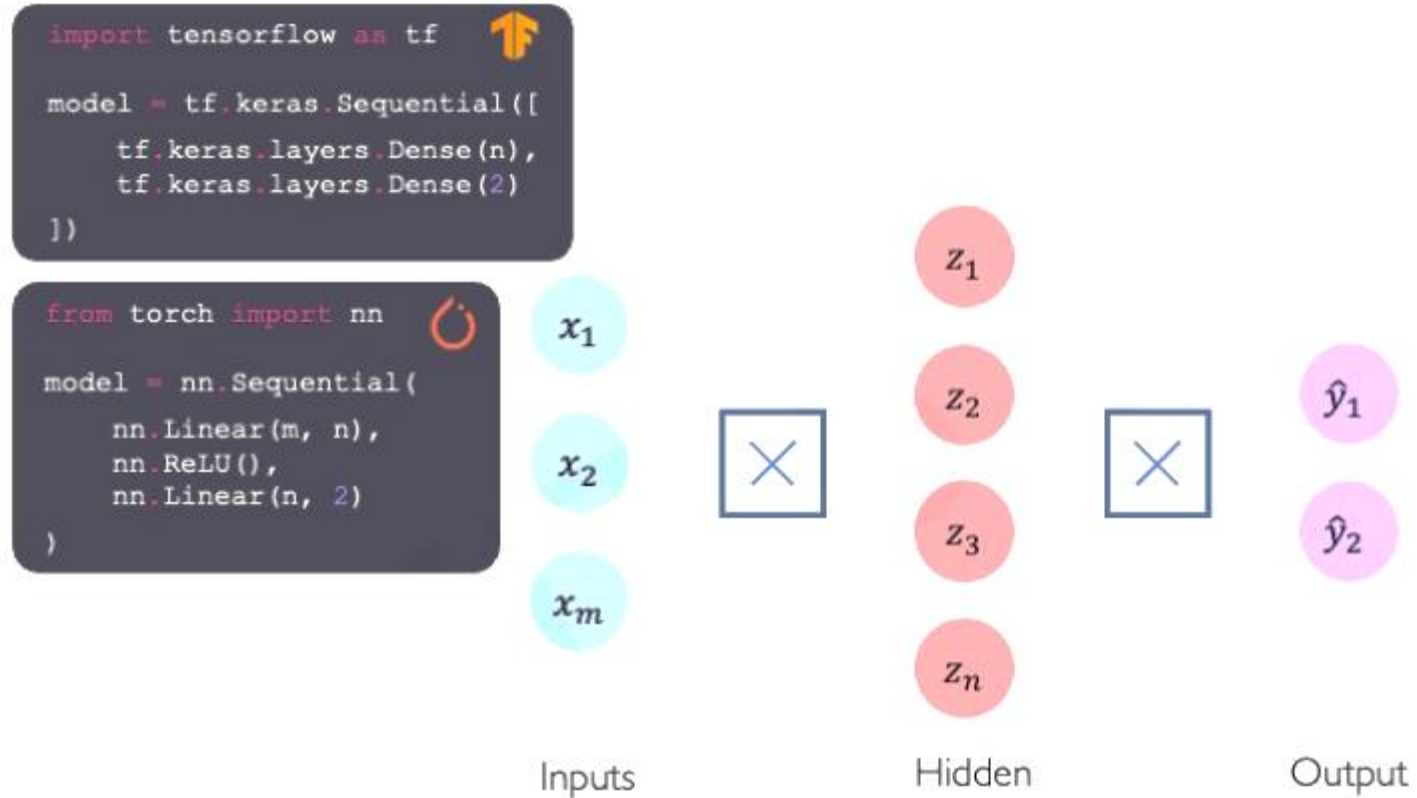
Red Neuronal de Una Sola Capa



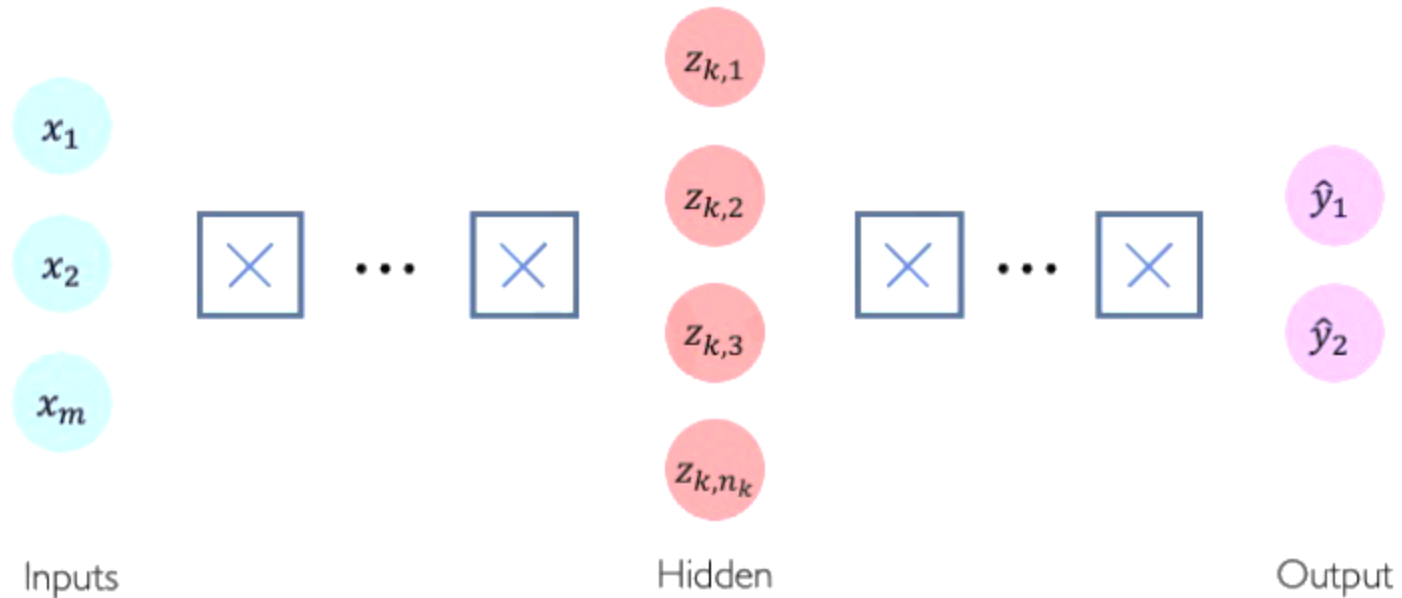
$$\begin{aligned} z_2 &= w_{0,2}^{(1)} + \sum_{j=1}^m x_j w_{j,2}^{(1)} \\ &= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)} \end{aligned}$$



Perceptrón de Salida Múltiple



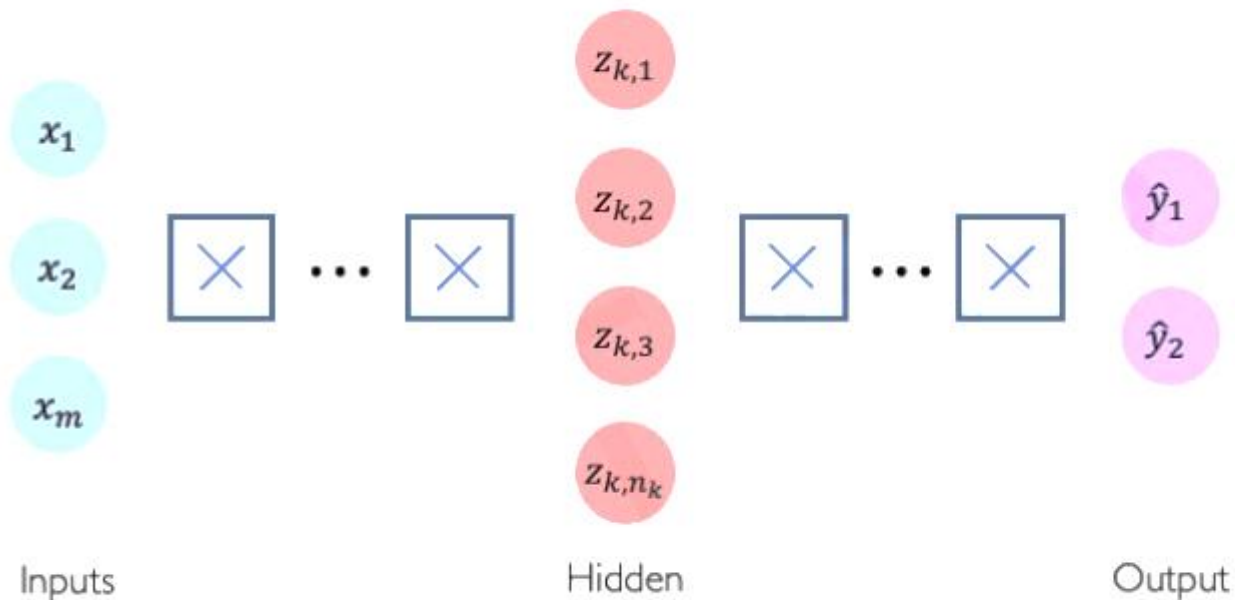
Red Neuronal Profunda



$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)}$$



Red Neuronal Profunda



$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)}$$

```
import tensorflow as tf

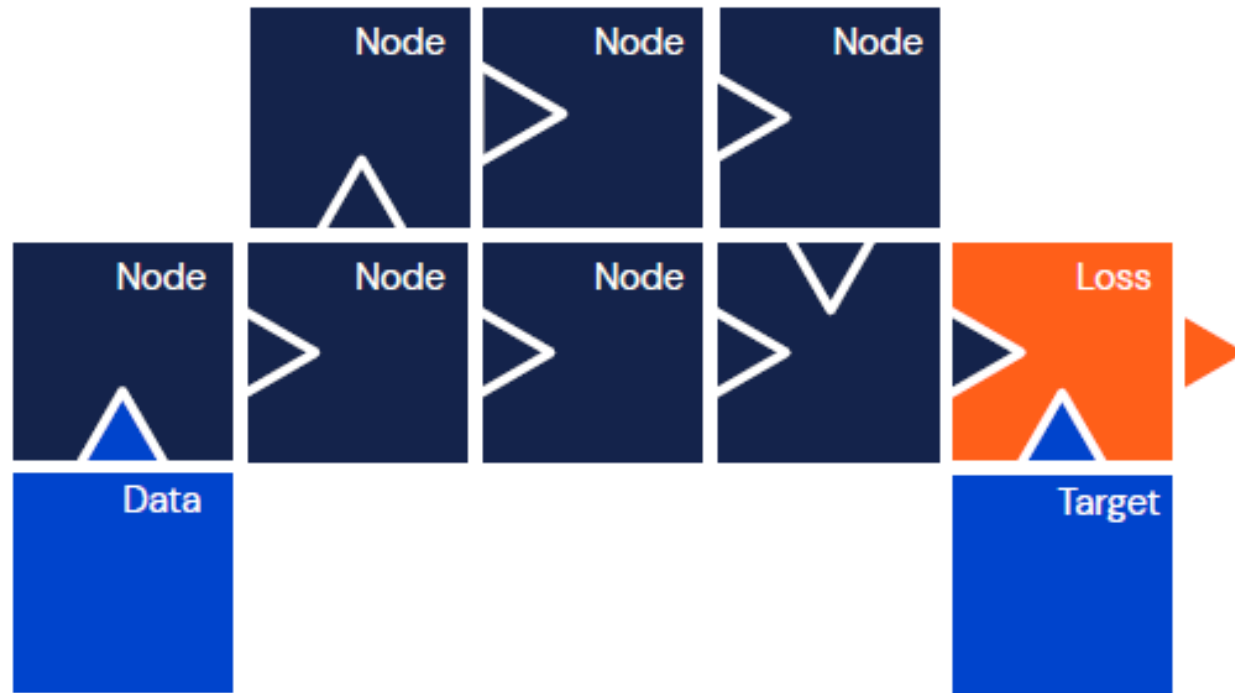
model = tf.keras.Sequential([
    tf.keras.layers.Dense(n1),
    tf.keras.layers.Dense(n2),
    :
    tf.keras.layers.Dense(2)
])
```

```
from torch import nn

model = nn.Sequential(
    nn.Linear(m, n1),
    nn.ReLU(),
    :
    nn.ReLU(),
    nn.Linear(nK, 2)
)
```



Deep learning – Lego blocks



Yann LeCun
@ylecun

Some folks still seem confused about what deep learning is. Here is a definition:

DL is constructing networks of parameterized functional modules & training them from examples using gradient-based optimization....
[facebook.com/722677142/post...](https://www.facebook.com/722677142/post...)

3:32 PM · Dec 24, 2019 · Facebook

517 Retweets 1.9K Likes



Danilo J. Rezende
@DeepSpiker

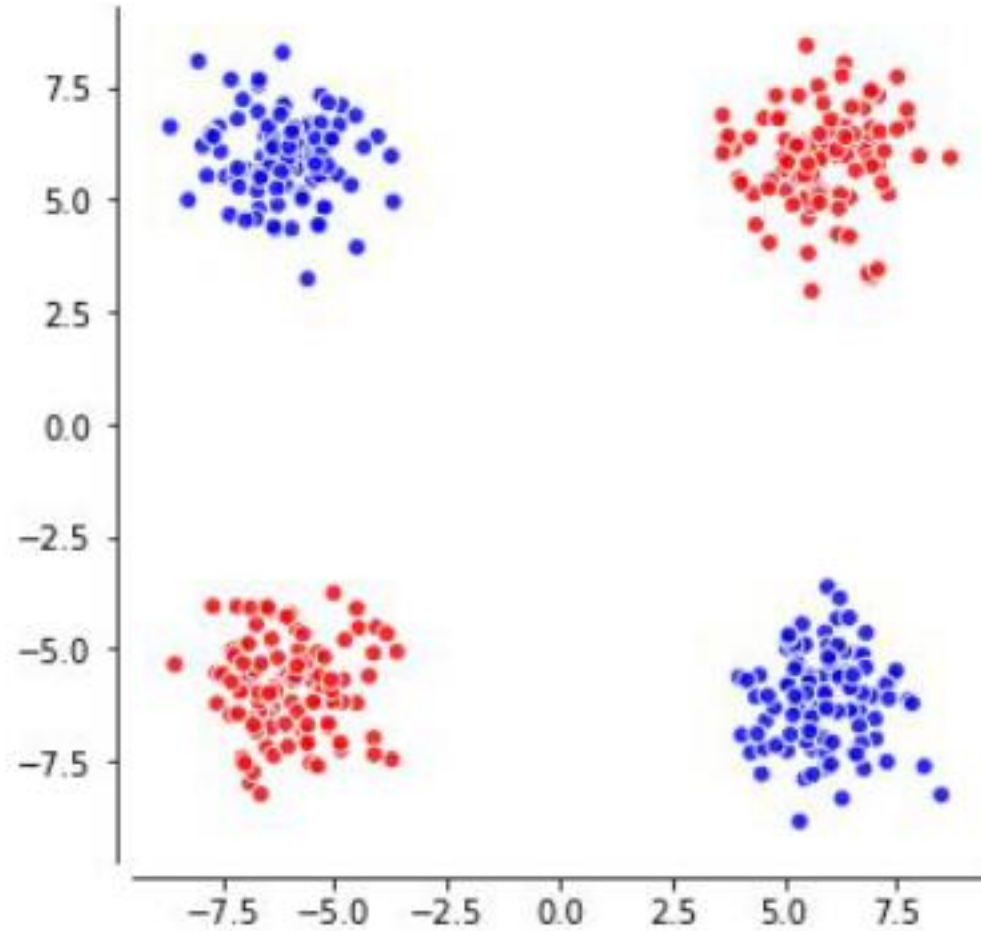
Rephrasing @ylecun with my own words: DL is a collection of tools to build complex modular differentiable functions. These tools are devoid of meaning, it is pointless to discuss what DL can or cannot do. What gives meaning to it is how it is trained and how the data is fed to it

3:43 PM · Dec 25, 2019 · Twitter for iPhone

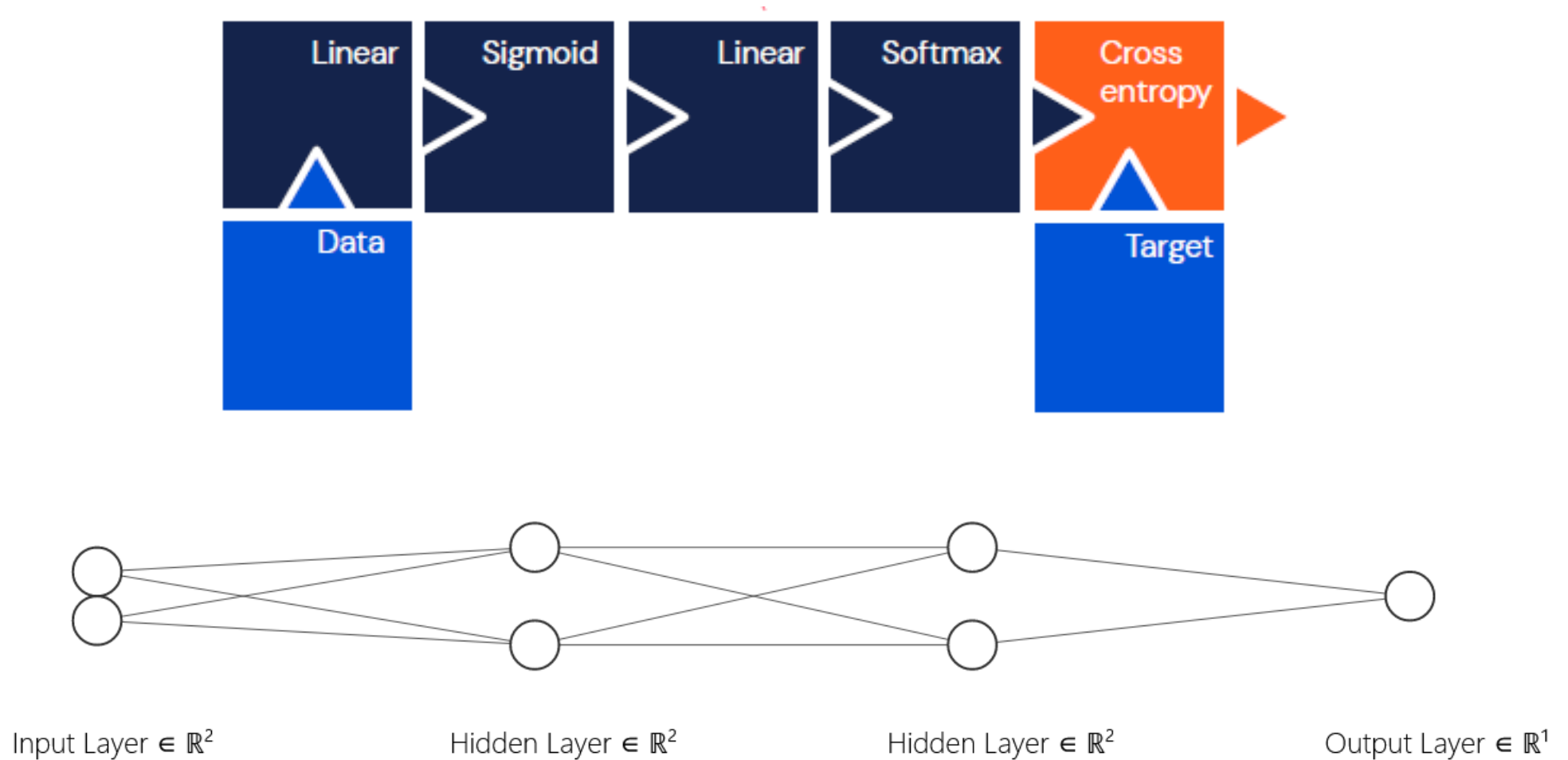
90 Retweets 464 Likes



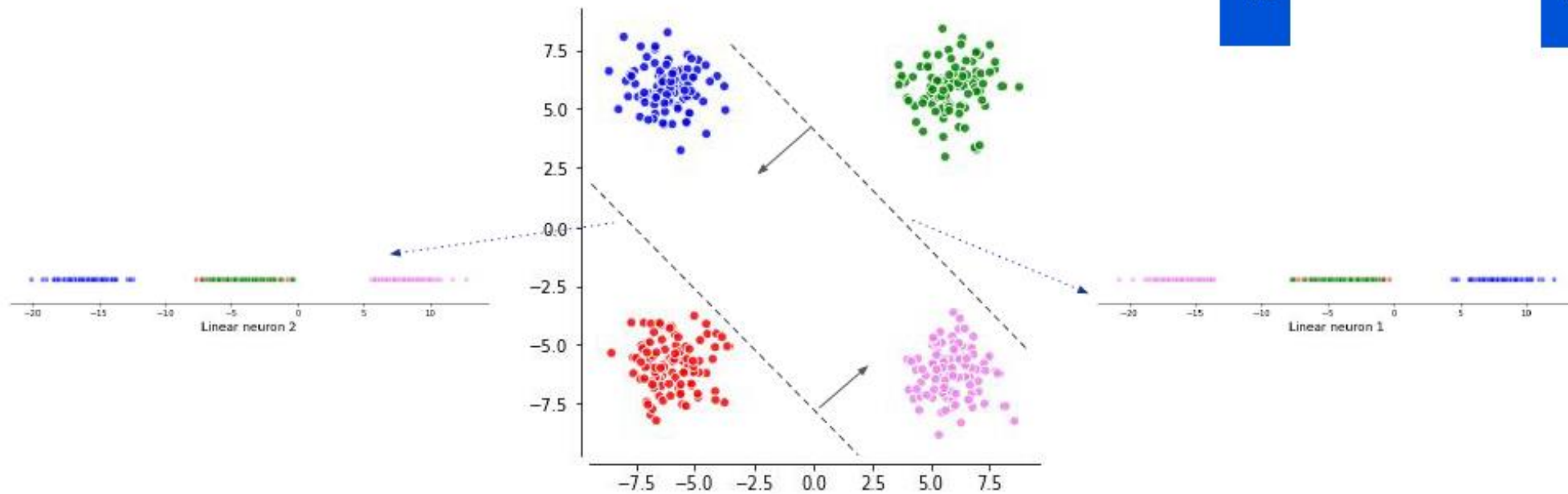
No linealidades



Red neuronal de dos capas

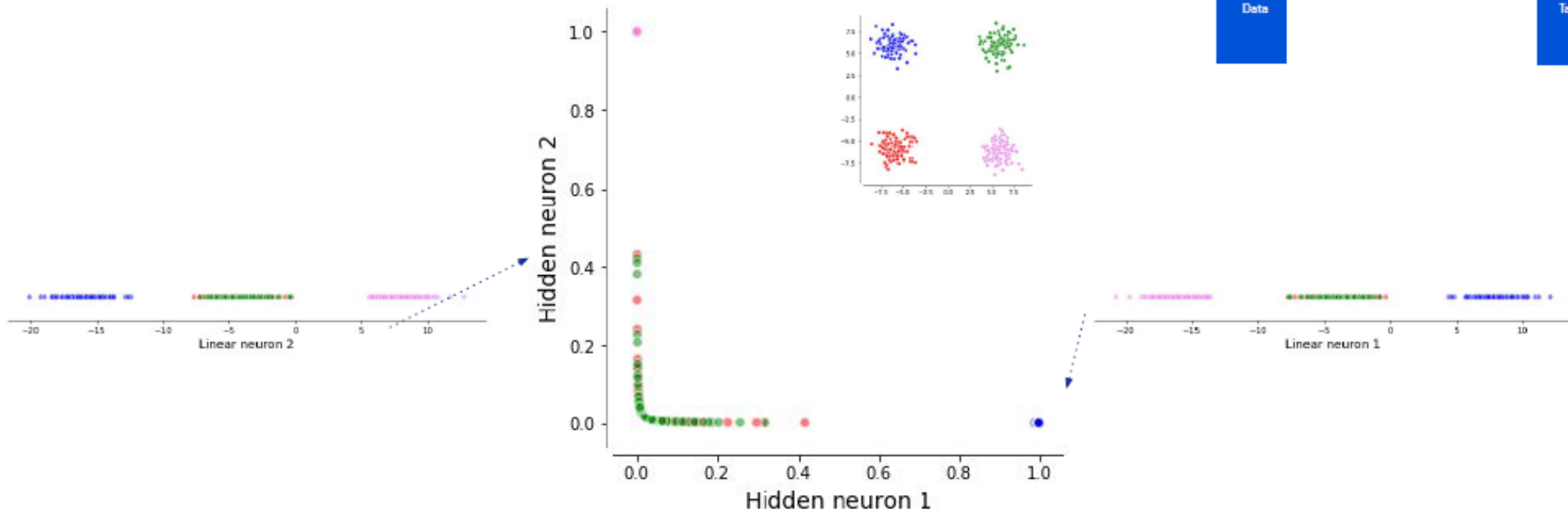


Red neuronal de dos capas



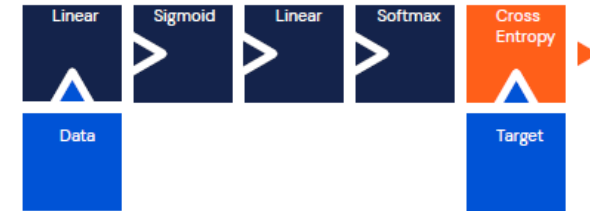
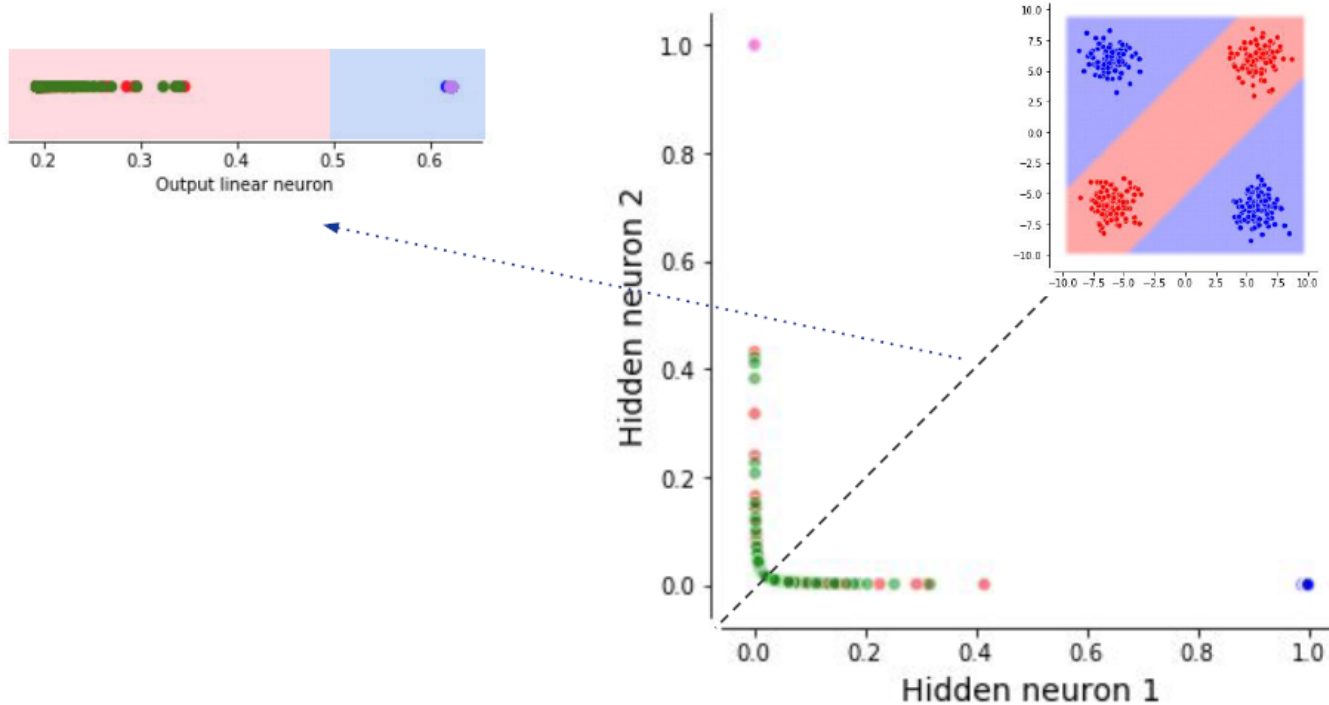
$$W = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \quad b = \begin{bmatrix} -4 \\ -4 \end{bmatrix}$$

Red neuronal de dos capas

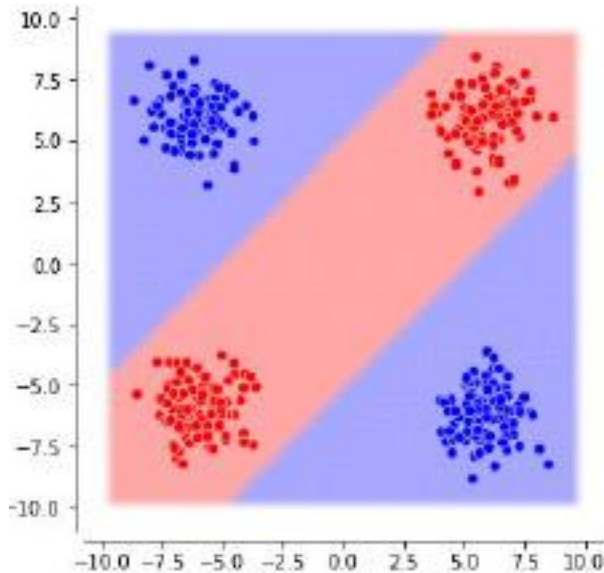


$$\mathbf{W} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -4 \\ -4 \end{bmatrix}$$

Red neuronal de dos capas



$$\mathbf{W} = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -4 \\ -4 \end{bmatrix}$$



$$W = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \quad b = \begin{bmatrix} -4 \\ -4 \end{bmatrix}$$

Las capas ocultas hacen transformaciones no-lineales en los datos de tal forma que una capa lineal al final pueda resolver el problema de clasificación

Aplicando Redes Neuronales



Problema de Ejemplo

¿Aprobaré esta clase?

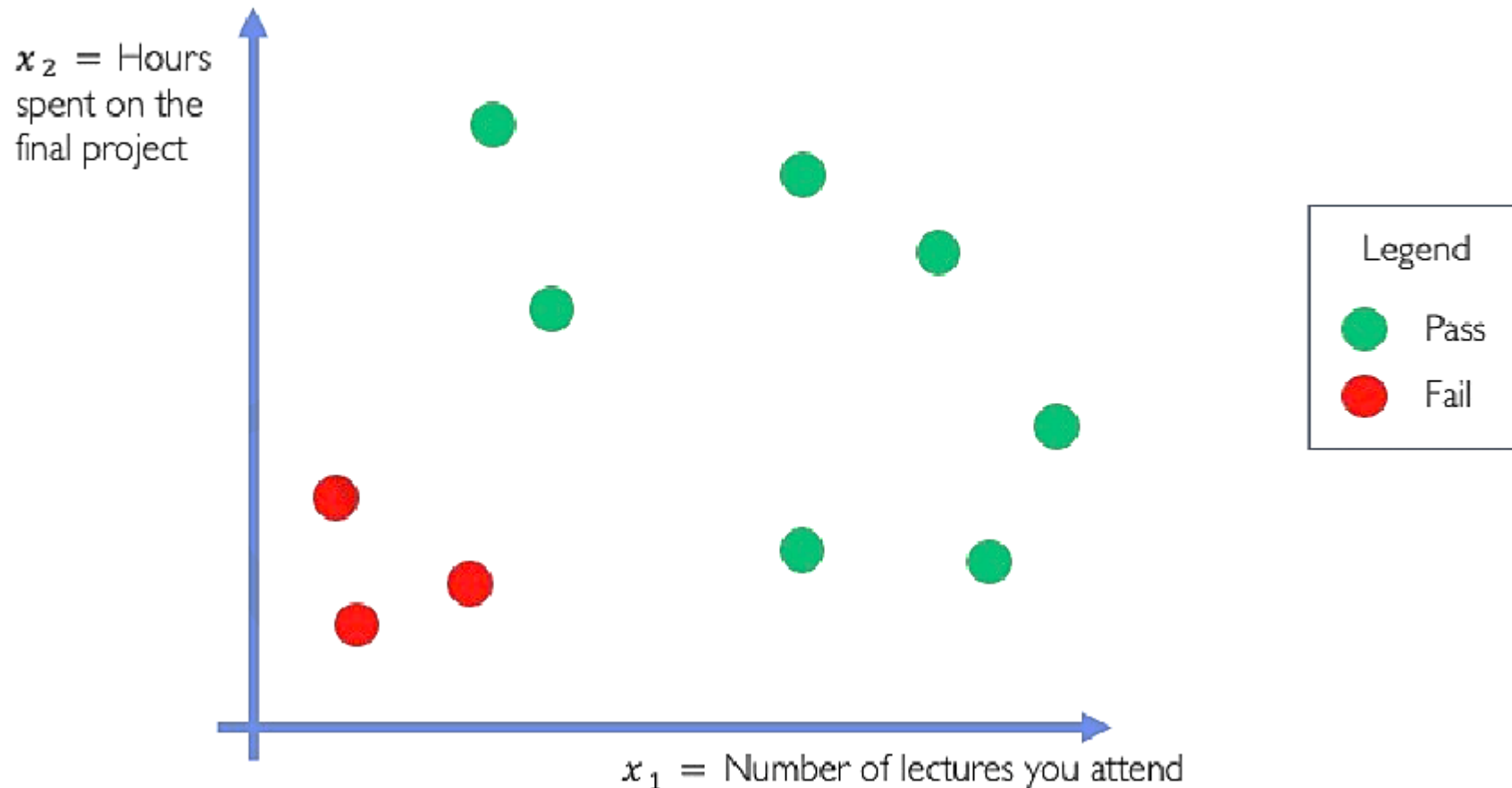
Comencemos con un modelo simple de dos características:

x_1 = Número de clases a las que asistes

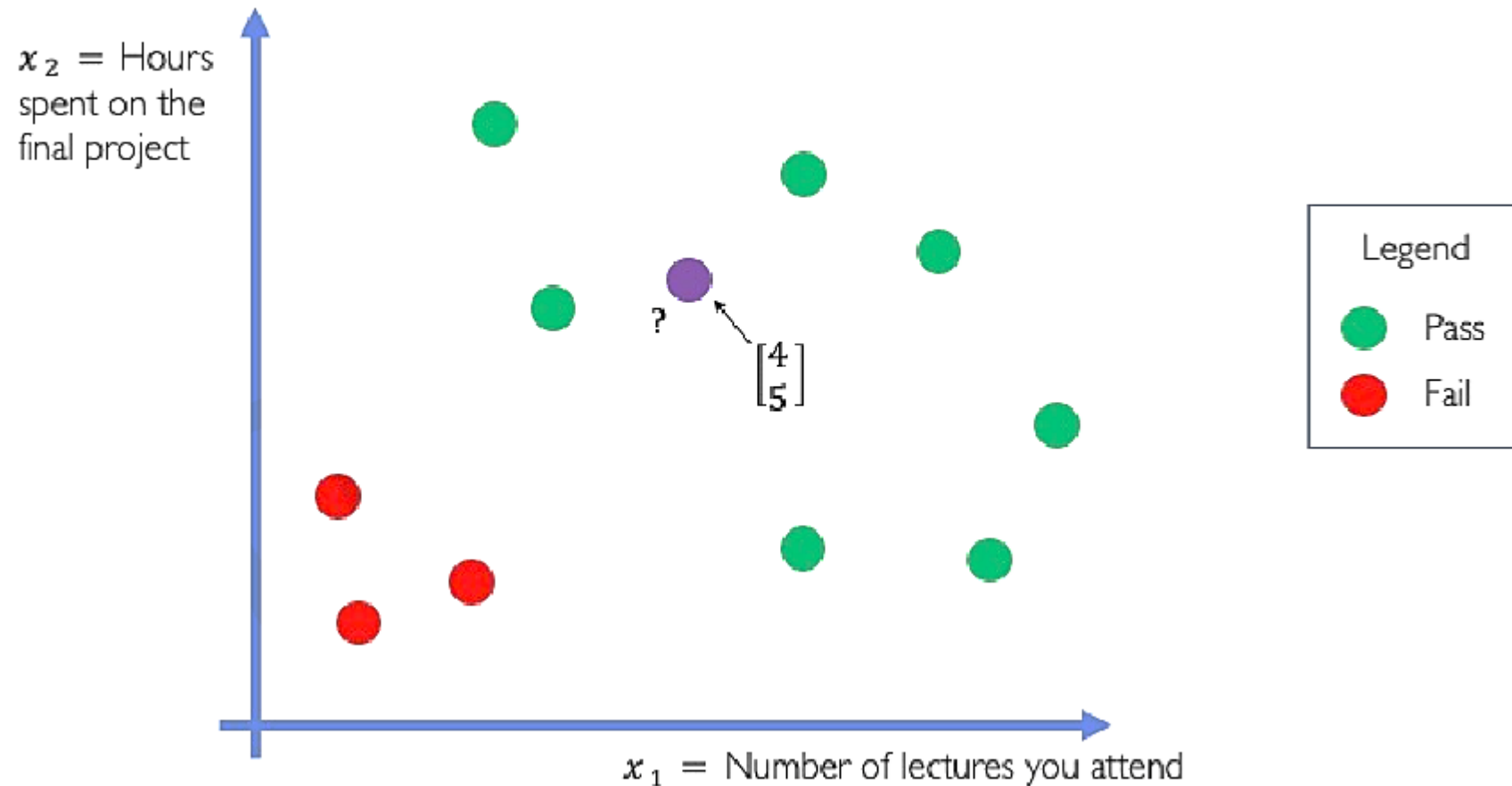
x_2 = Horas dedicadas al proyecto final



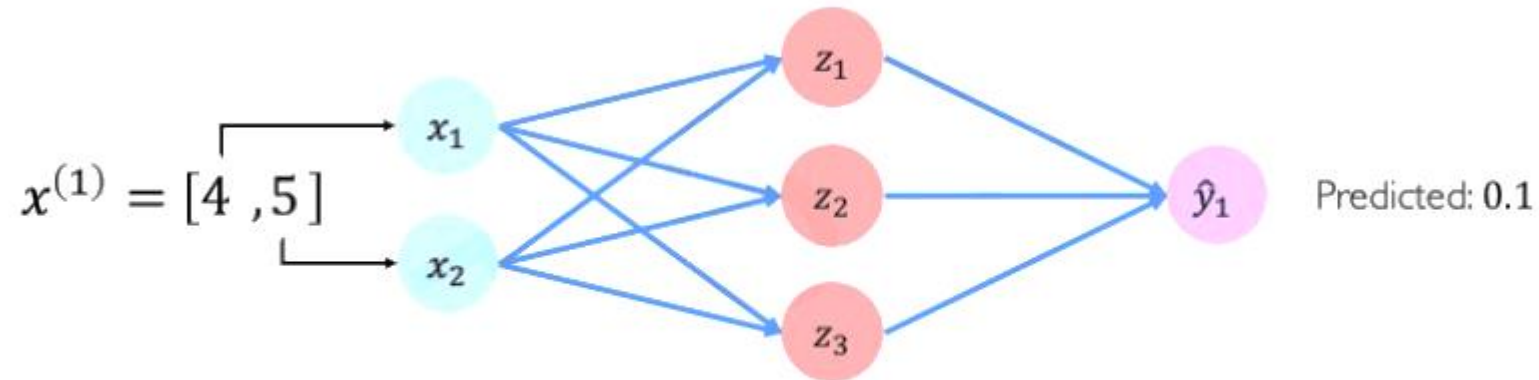
Problema de Ejemplo: ¿Aprobaré esta clase?



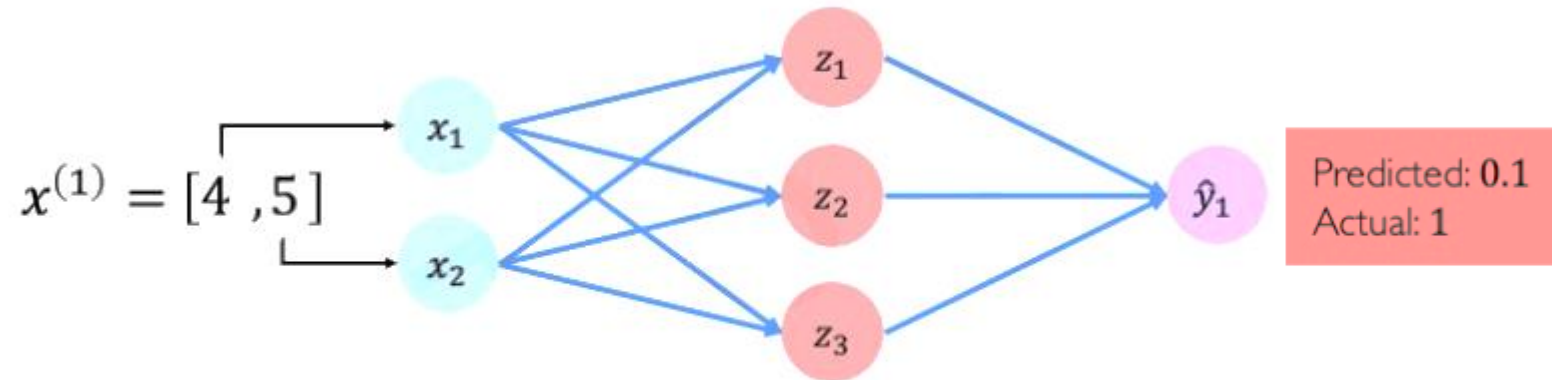
Problema de Ejemplo: ¿Aprobaré esta clase?



Problema de Ejemplo: ¿Aprobaré esta clase?

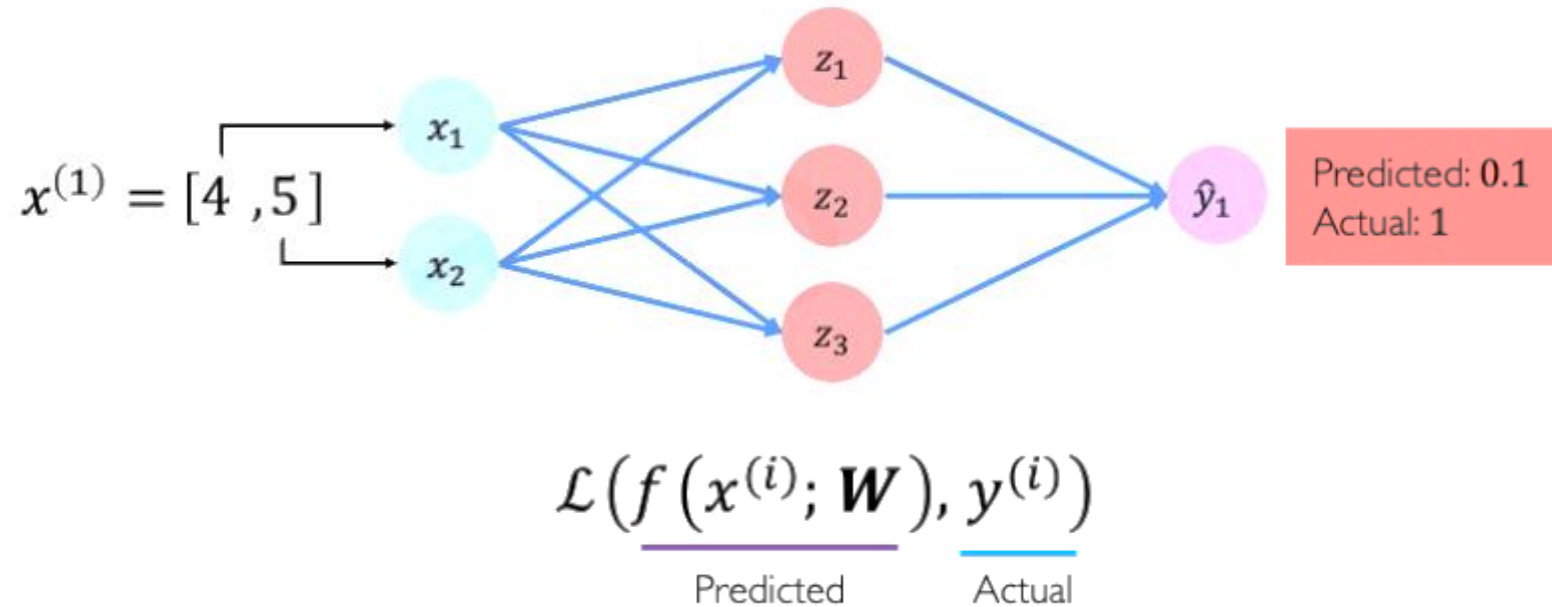


Problema de Ejemplo: ¿Aprobaré esta clase?



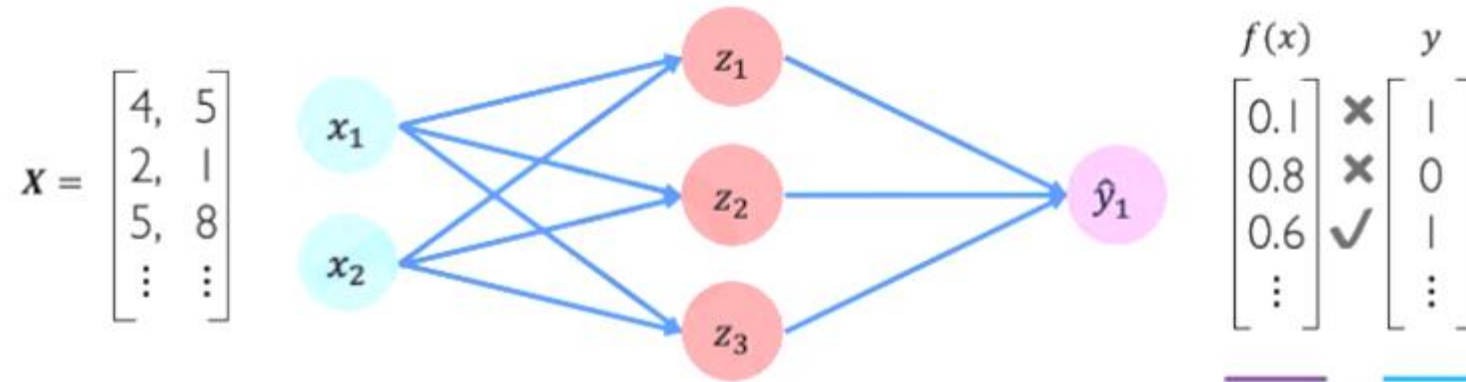
Cuantificando la Pérdida

La **pérdida** de nuestra red mide el costo incurrido por predicciones incorrectas.



Pérdida Empírica

La pérdida empírica mide la pérdida total sobre todo nuestro conjunto de datos.



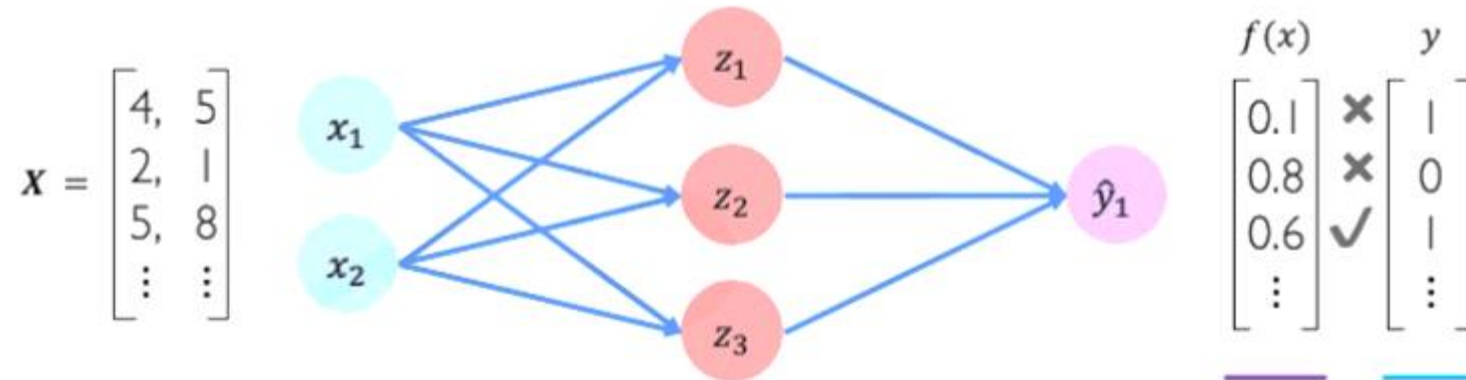
- Also known as:
- Objective function
 - Cost function
 - Empirical Risk

$$J(W) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\underbrace{f(x^{(i)}; W)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$



Pérdida por Entropía Cruzada Binaria

La pérdida por entropía cruzada puede utilizarse con modelos que generan una probabilidad entre 0 y 1.



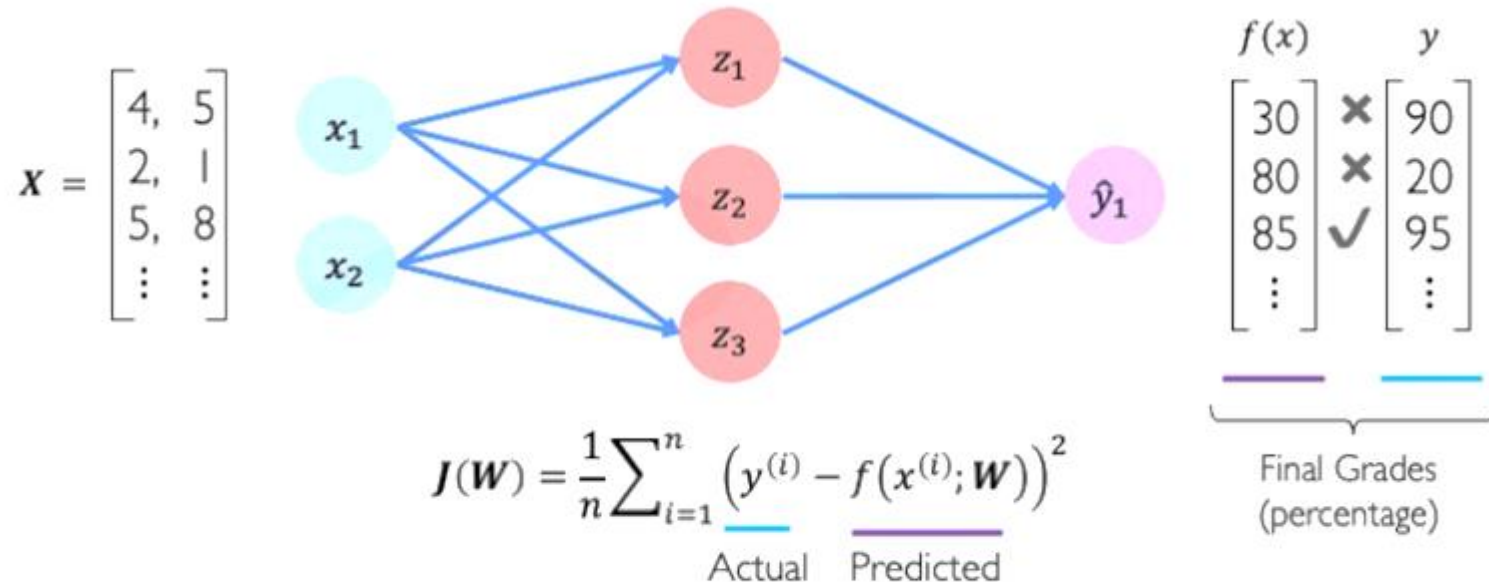
$$J(W) = -\frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)}}_{\text{Actual}} \log \left(\underbrace{f(x^{(i)}; W)}_{\text{Predicted}} \right) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log \left(1 - \underbrace{f(x^{(i)}; W)}_{\text{Predicted}} \right)$$

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )  
loss = torch.nn.functional.cross_entropy( predicted, y )
```



Pérdida de Error Cuadrático Medio (Mean Squared Error Loss)

La pérdida por error cuadrático medio puede utilizarse con modelos de regresión que generan números reales continuos



```
loss = tf.reduce_mean( tf.square( tf.subtract( y, predicted ) ) )  
loss = tf.keras.losses.MSE( y, predicted )
```

```
loss = torch.nn.functional.mse_loss( predicted, y )
```



Entrenando Redes Neuronales



Optimización de la Pérdida

Queremos encontrar los pesos de la red que logren la menor pérdida posible

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$



Optimización de la Pérdida

Queremos encontrar los pesos de la red que logren la menor pérdida posible

$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} J(\mathbf{W})$$

Remember:

$$\mathbf{W} = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, \dots\}$$

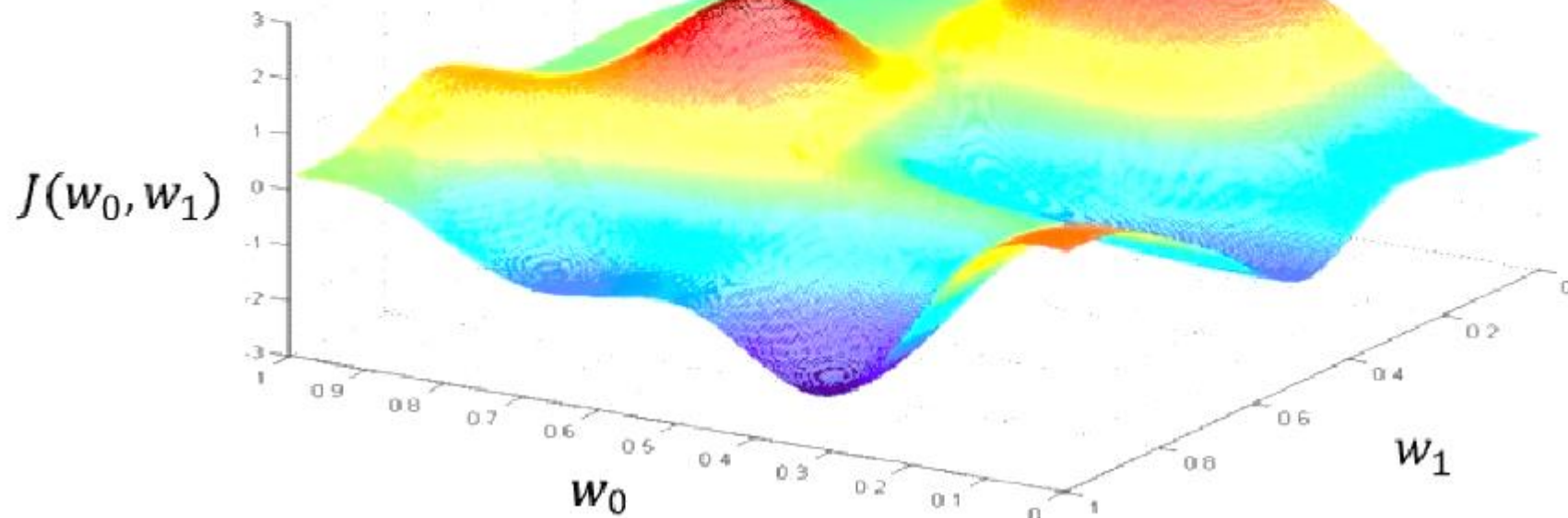


Optimización de la Pérdida

$$W^* = \underset{W}{\operatorname{argmin}} J(W)$$

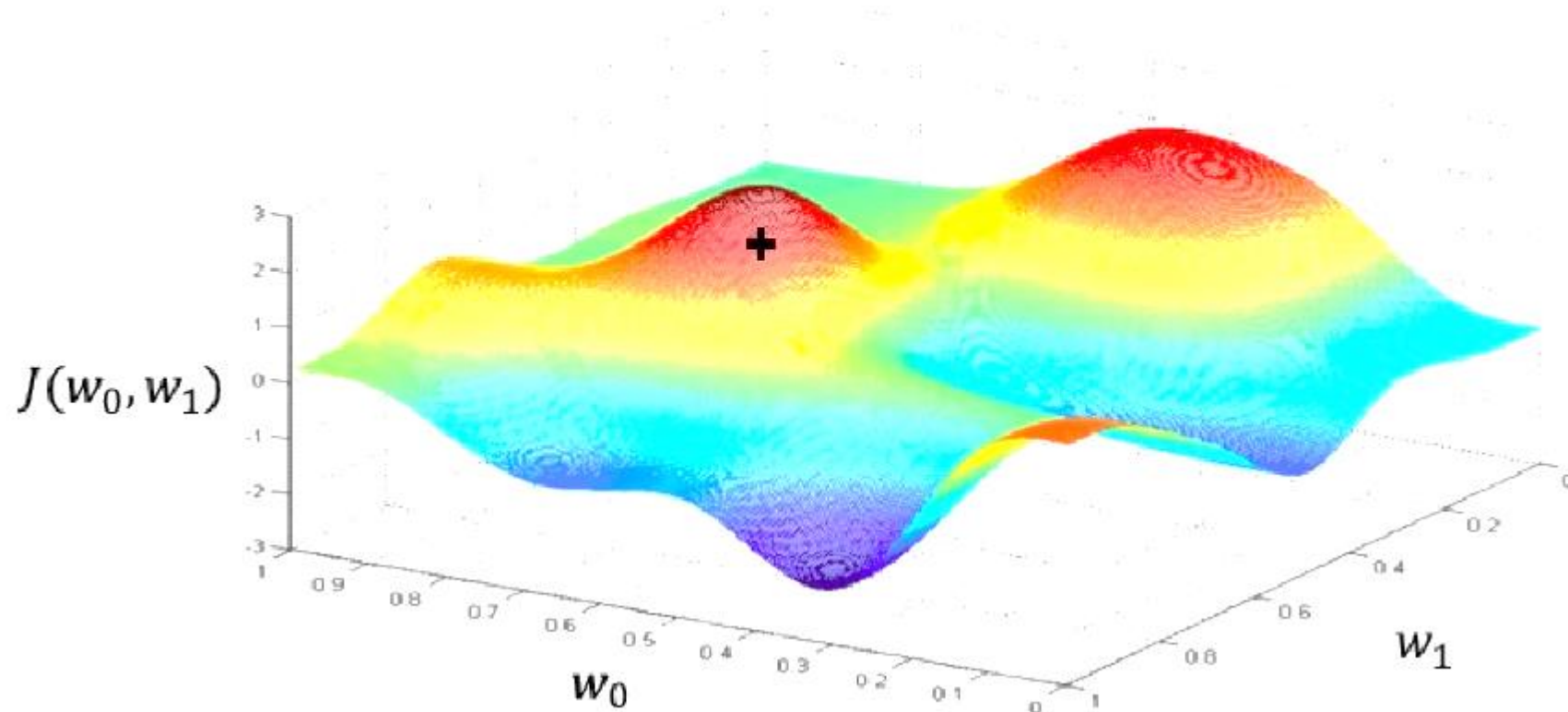
Recuerda:

¡Nuestra función de
pérdida depende de los
pesos de la red!



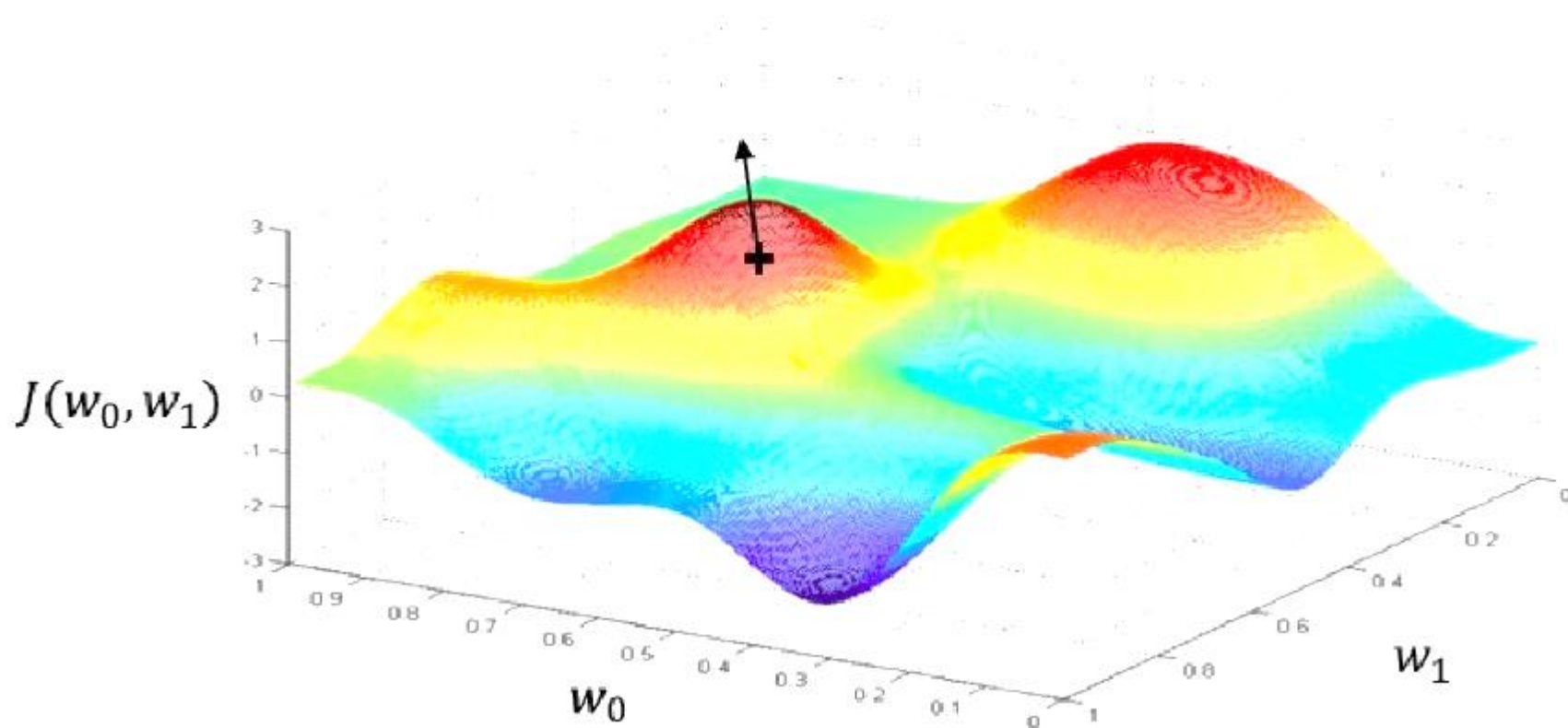
Optimización de la Pérdida

Selecciona aleatoriamente un punto (w_0, w_1)
inicial



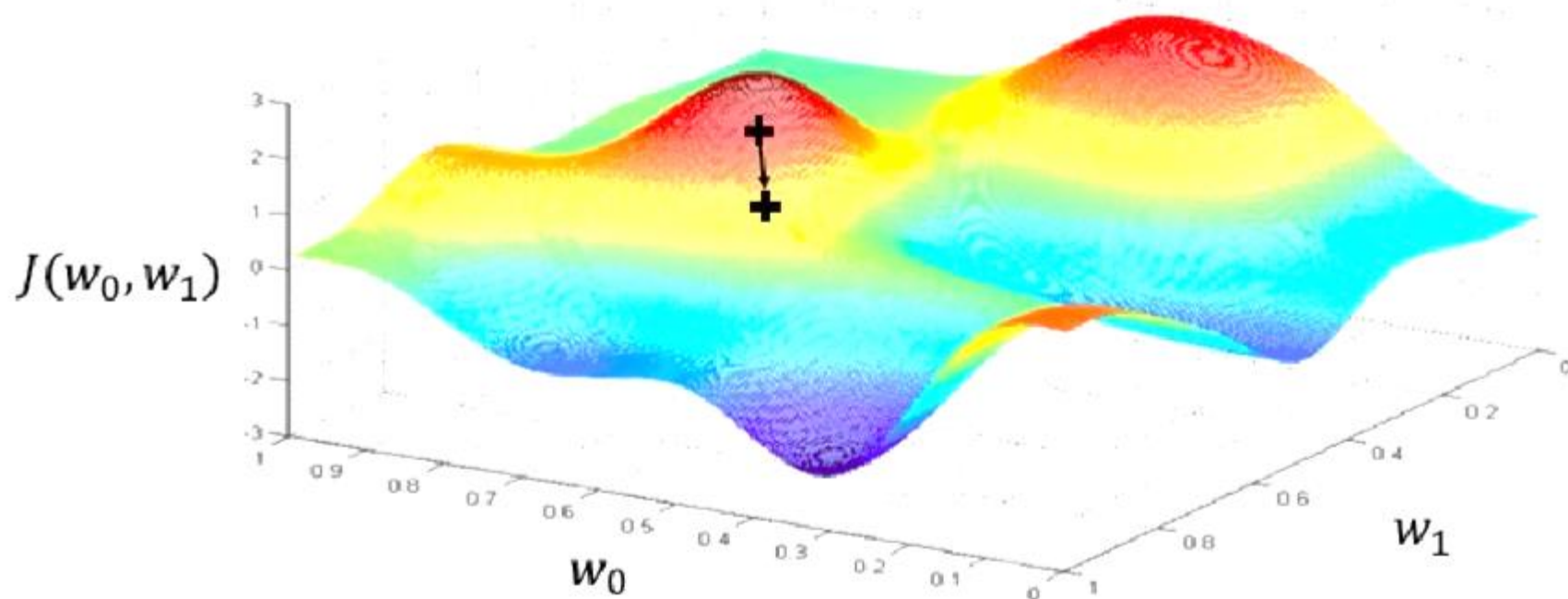
Optimización de la Pérdida

Calcula el gradiente, $\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$



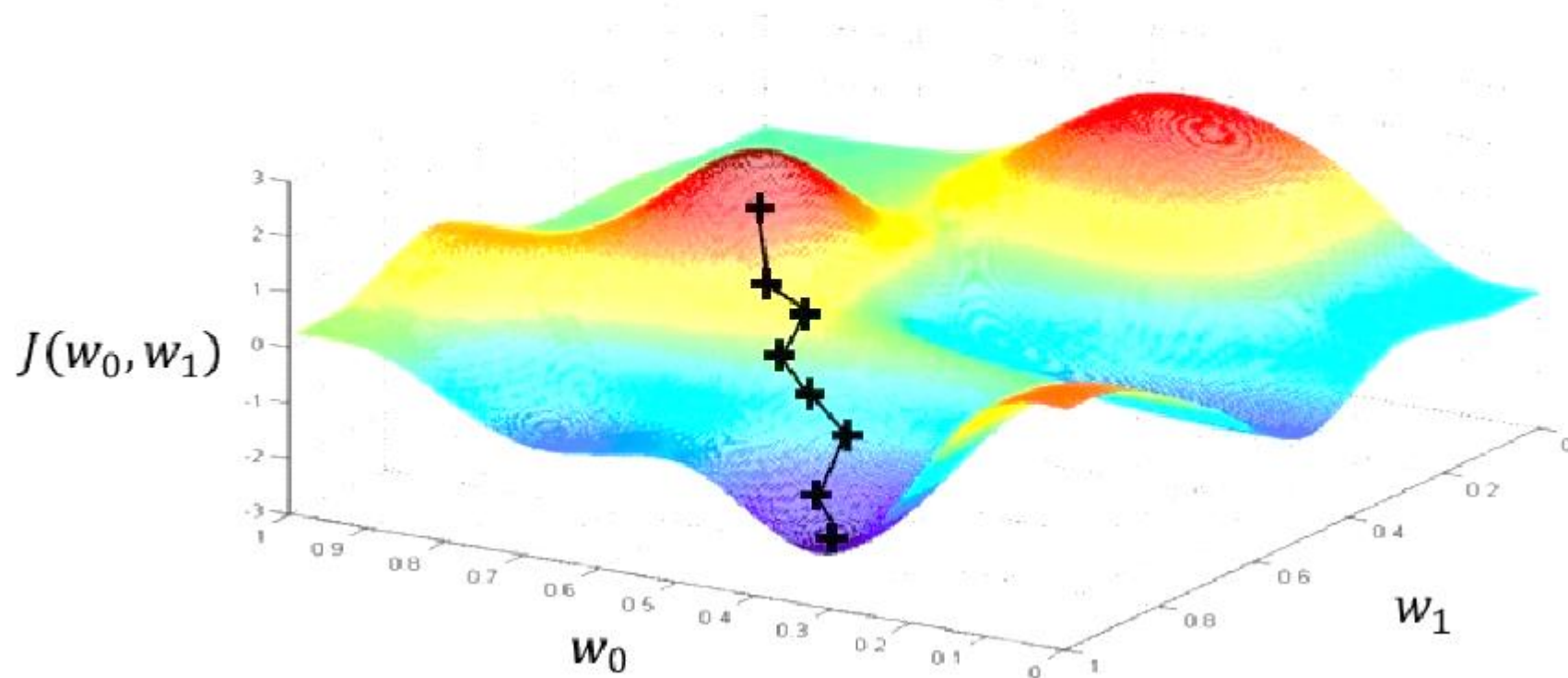
Optimización de la Pérdida

Da un pequeño paso en la dirección opuesta al gradiente



Descenso por Gradiente

Repetir hasta la convergencia

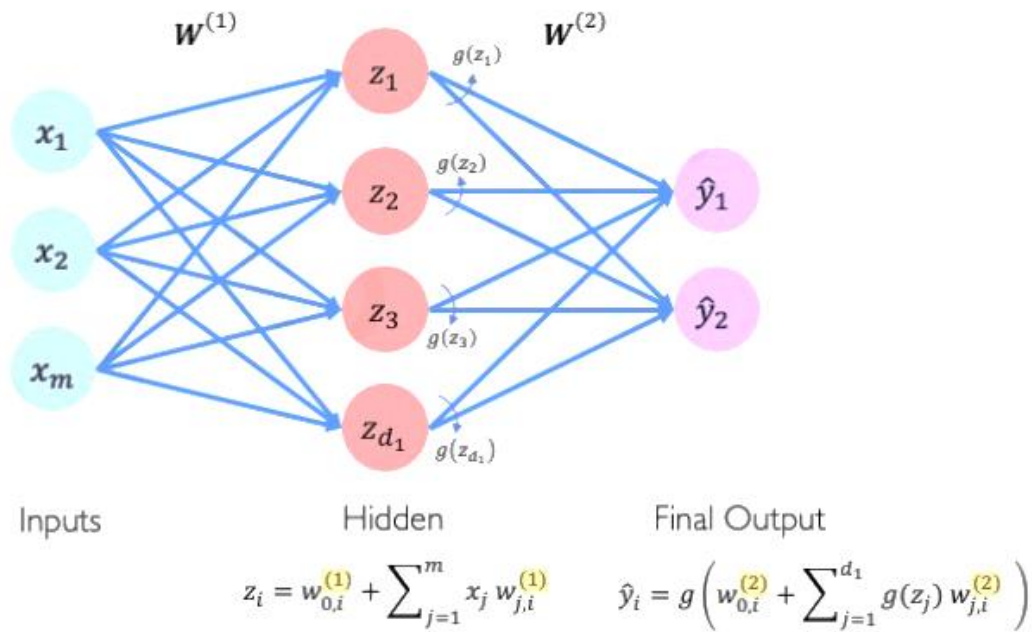


Algoritmo

1. Inicializar los pesos aleatoriamente $\sim \mathcal{N}(0, \sigma^2)$
2. Repetir hasta la convergencia:
3. Calcular el gradiente, $\frac{\partial J(W)}{\partial W}$
4. Actualizar los pesos, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Devolver los pesos




Derivadas parciales



Descenso por Gradiente

Algoritmo

1. Inicializar los pesos aleatoriamente $\sim \mathcal{N}(0, \sigma^2)$
2. Repetir hasta la convergencia:
3. Calcular el gradiente, $\frac{\partial J(W)}{\partial W}$
4. Actualizar los pesos, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Devolver los pesos



```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)


    weights = weights - lr * gradient
```



Descenso por Gradiente

Algoritmo

1. Inicializar los pesos aleatoriamente $\sim \mathcal{N}(0, \sigma^2)$
2. Repetir hasta la convergencia:
3. Calcular el gradiente, $\frac{\partial J(W)}{\partial W}$
4. Actualizar los pesos, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Devolver los pesos



```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```



Cálculo de Gradientes: Retropropagación



¿Cómo afecta un pequeño cambio en un peso (ej. w_2) a la pérdida final $J(W)$?



Cálculo de Gradientes: Retropropagación

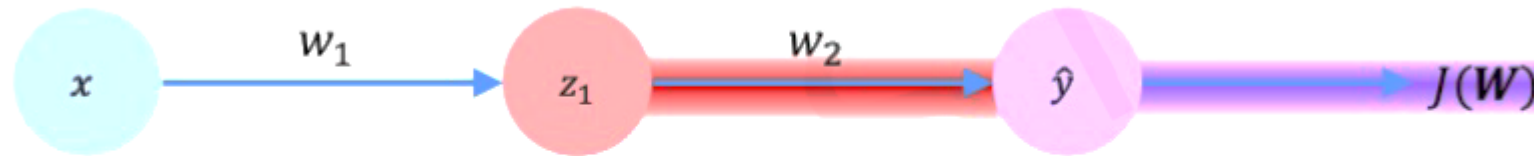


$$\frac{\partial J(W)}{\partial w_2} =$$

¡Usemos la regla de la cadena!



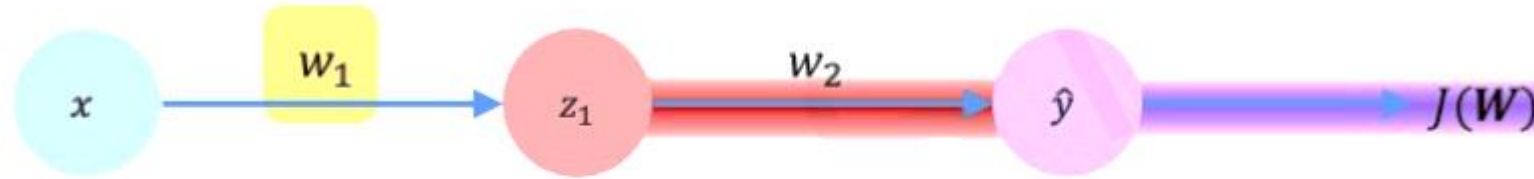
Cálculo de Gradientes: Retropropagación



$$\frac{\partial J(W)}{\partial w_2} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial w_2}}_{\text{red}}$$



Cálculo de Gradientes: Retropropagación



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

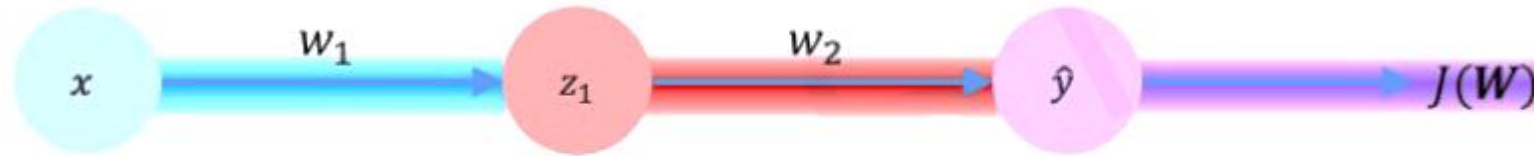
The diagram shows the chain rule for the gradient of the loss function $J(W)$ with respect to the weight w_1 . The first term, $\frac{\partial J(W)}{\partial w_1}$, is highlighted with a yellow box. The second term, $\frac{\partial J(W)}{\partial \hat{y}}$, is highlighted with a purple box. The third term, $\frac{\partial \hat{y}}{\partial w_1}$, is highlighted with a red box. Arrows point from the text below to these terms.

¡Usemos la regla de la cadena!

¡Usemos la regla de la cadena!



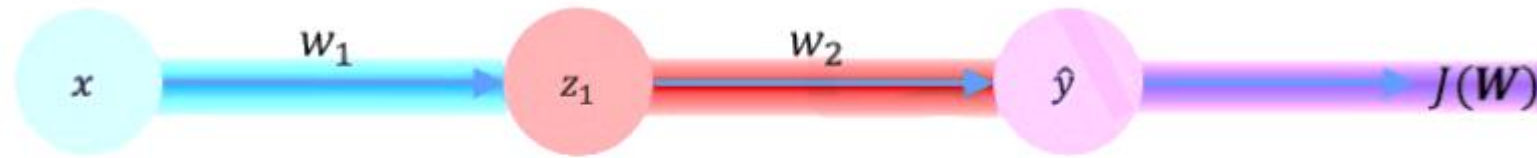
Cálculo de Gradientes: Retropropagación



$$\frac{\partial J(W)}{\partial w_1} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$



Cálculo de Gradientes: Retropropagación



$$\frac{\partial J(W)}{\partial w_1} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

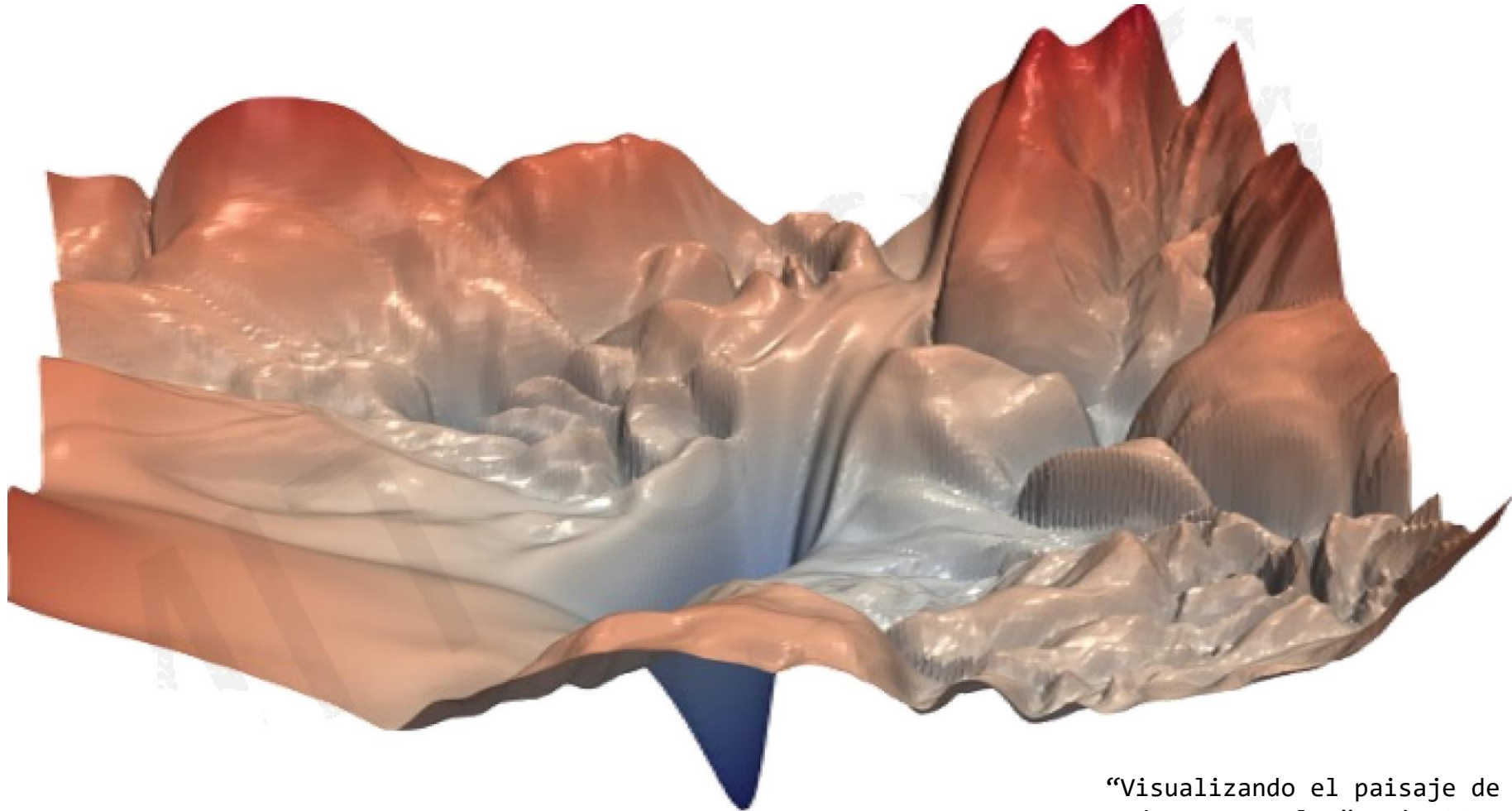
Repita esto para cada peso en la red usando gradientes de las capas posteriores



Redes Neuronales en la Práctica: Optimización



"Entrenar redes neuronales es difícil"



“Visualizando el paisaje de pérdida de las redes neuronales”. Dic 2017.



Las funciones de pérdida pueden ser difíciles de optimizar

Recuerda:

Optimización mediante descenso por gradiente

$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$



Las funciones de pérdida pueden ser difíciles de optimizar

Recuerda:

Optimización mediante descenso por gradiente

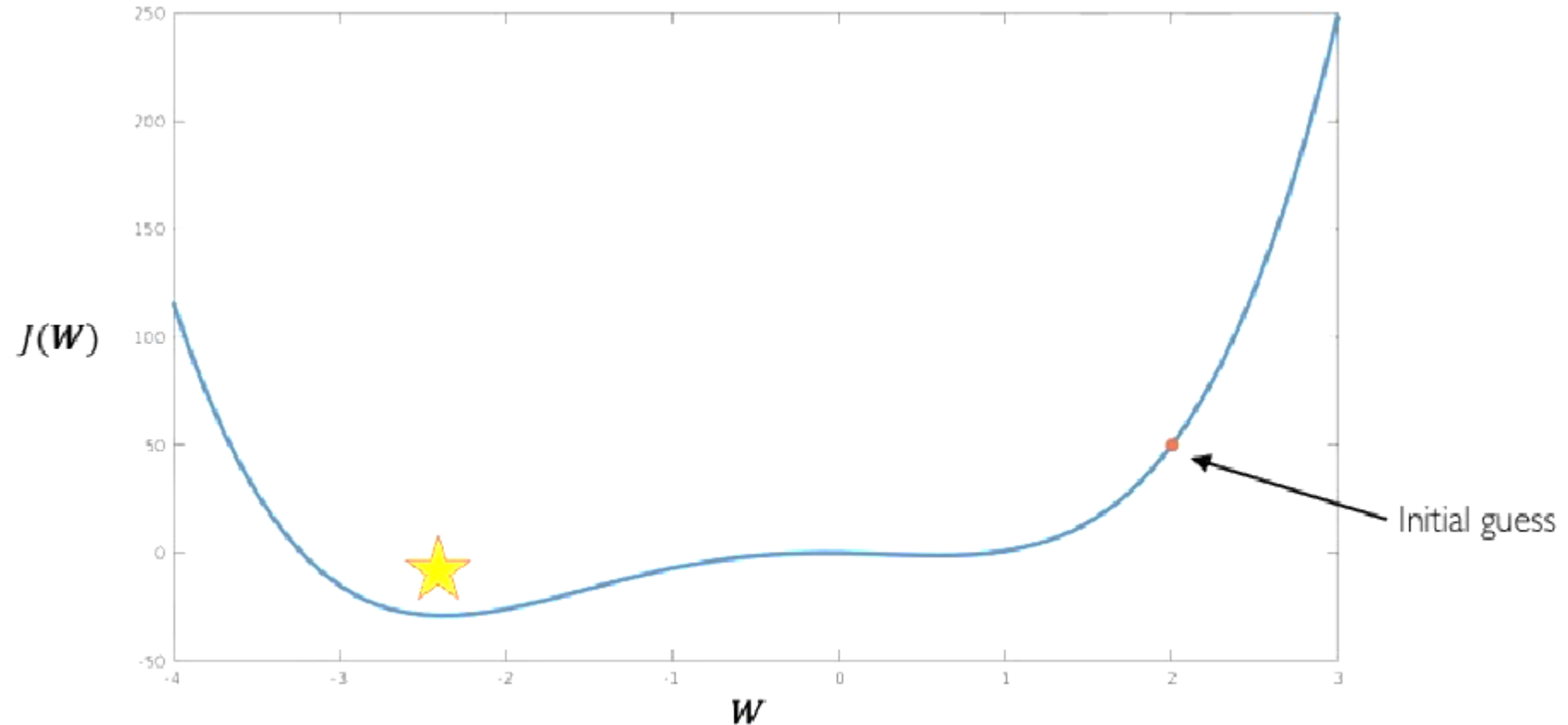
$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$

How can we set the
learning rate?



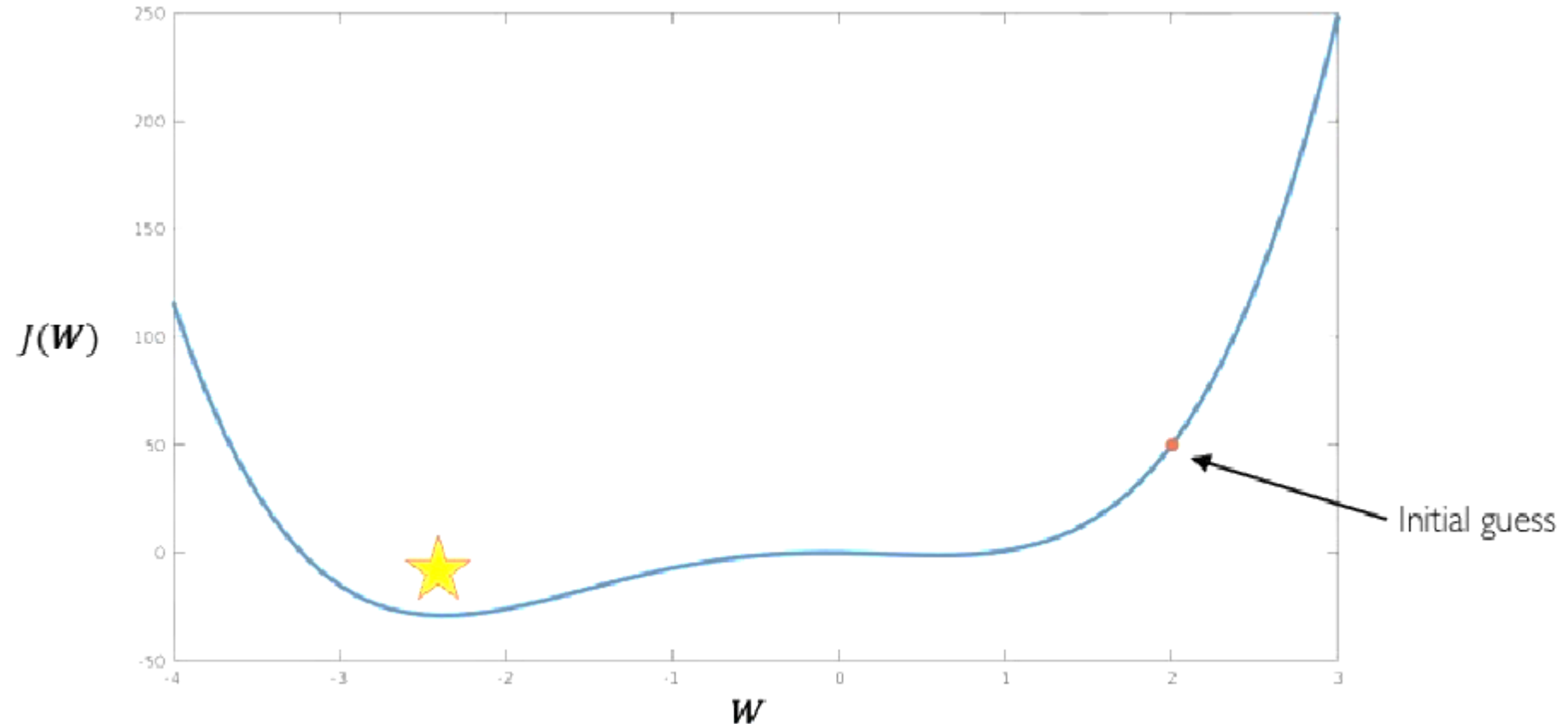
Estableciendo la tasa de aprendizaje

Una tasa de aprendizaje pequeña converge lentamente y queda atrapada en mínimos locales falsos



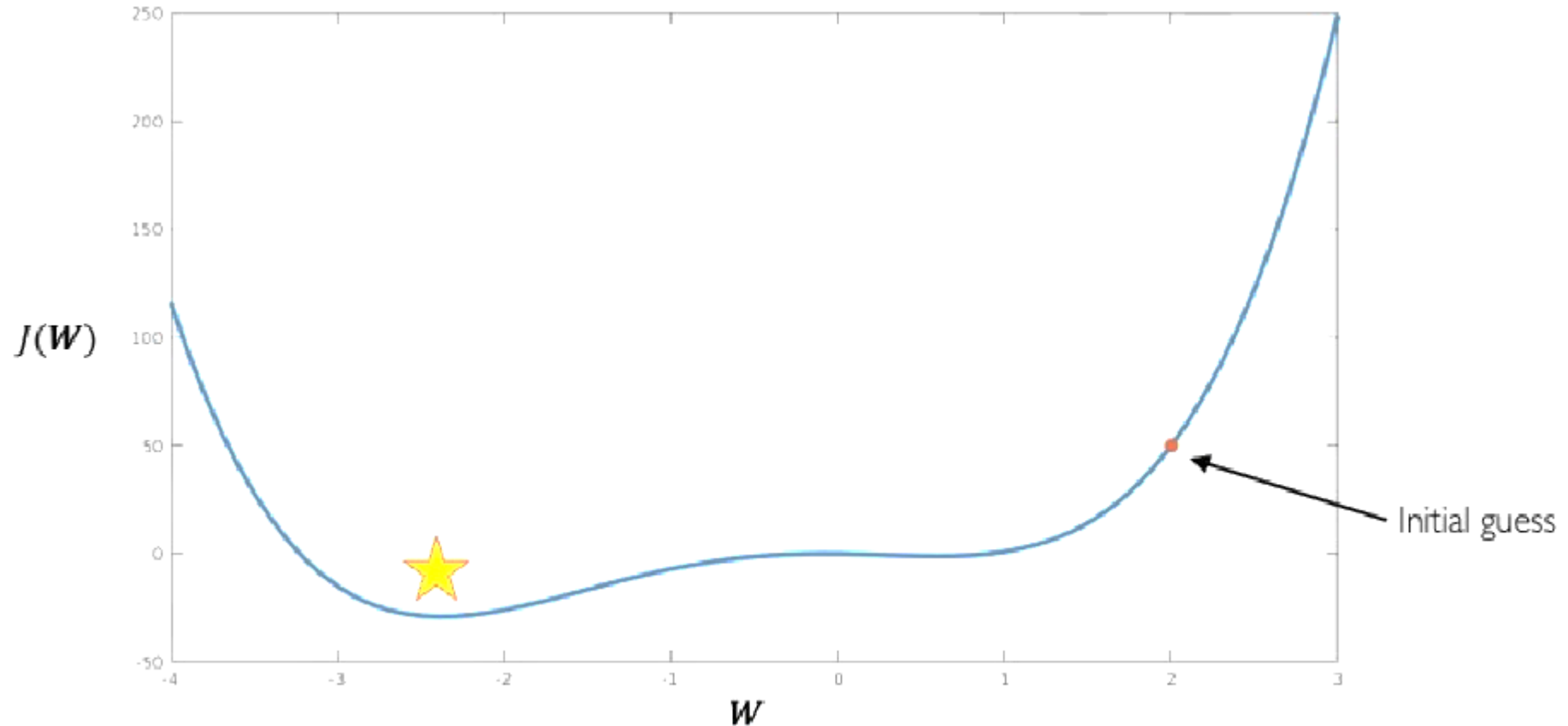
Estableciendo la tasa de aprendizaje

Tasas de aprendizaje grandes se exceden, se vuelven inestables y divergen.



Estableciendo la tasa de aprendizaje

Las tasas de aprendizaje estables convergen suavemente y evitan mínimos locales



¿Cómo lidiar con esto?

Idea 1:

Probar muchas tasas de aprendizaje diferentes y ver cuál funciona “justo bien”

Idea 2:

¡Haz algo más inteligente!

Diseña una tasa de aprendizaje adaptativa que se “adapte” al paisaje



Tasas de aprendizaje adaptativas

- Las tasas de aprendizaje ya no son fijas
- Pueden hacerse más grandes o más pequeñas dependiendo de:
 - qué tan grande es el gradiente
 - qué tan rápido está ocurriendo el aprendizaje
 - el tamaño de ciertos pesos
 - etc...



Algoritmos de descenso por gradiente

Algorithm	TF Implementation	Torch Implementation	Reference
• SGD	 <code>tf.keras.optimizers.SGD</code>	 <code>torch.optim.SGD</code>	Kiefer & Wolfowitz, 1952.
• Adam	 <code>tf.keras.optimizers.Adam</code>	 <code>torch.optim.Adam</code>	Kingma et al., 2014.
• Adadelta	 <code>tf.keras.optimizers.Adadelta</code>	 <code>torch.optim.Adadelta</code>	Zeiler et al., 2012.
• Adagrad	 <code>tf.keras.optimizers.Adagrad</code>	 <code>torch.optim.Adagrad</code>	Duchi et al., 2011.
• RMSProp	 <code>tf.keras.optimizers.RMSProp</code>	 <code>torch.optim.RMSProp</code>	

Detalles adicionales: <http://ruder.io/optimizing-gradient-descent/>



Juntándolo todo

```
import tensorflow as tf

model = tf.keras.Sequential([...])

# pick your favorite optimizer
optimizer = tf.keras.optimizer.SGD()

while True: # loop forever

    # forward pass through the network
    prediction = model(x)

    with tf.GradientTape() as tape:
        # compute the loss
        loss = compute_loss(y, prediction)

    # update the weights using the gradient
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```



Can replace with
any TensorFlow
optimizer!



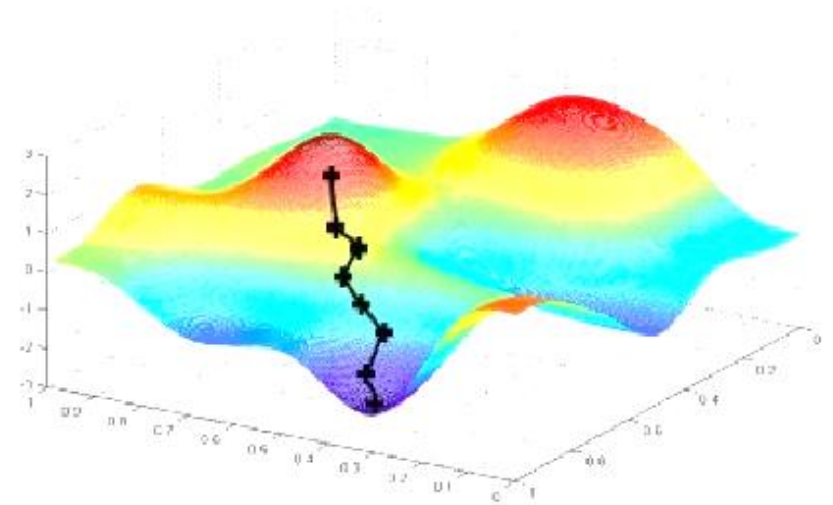
Redes neuronales en la práctica: Mini-lotes



Descenso por Gradiente

Algoritmo

1. Inicializar los pesos aleatoriamente $\sim \mathcal{N}(0, \sigma^2)$
2. Repetir hasta la convergencia:
3. Calcular el gradiente, $\frac{\partial J(W)}{\partial W}$
4. Actualizar los pesos, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Devolver los pesos

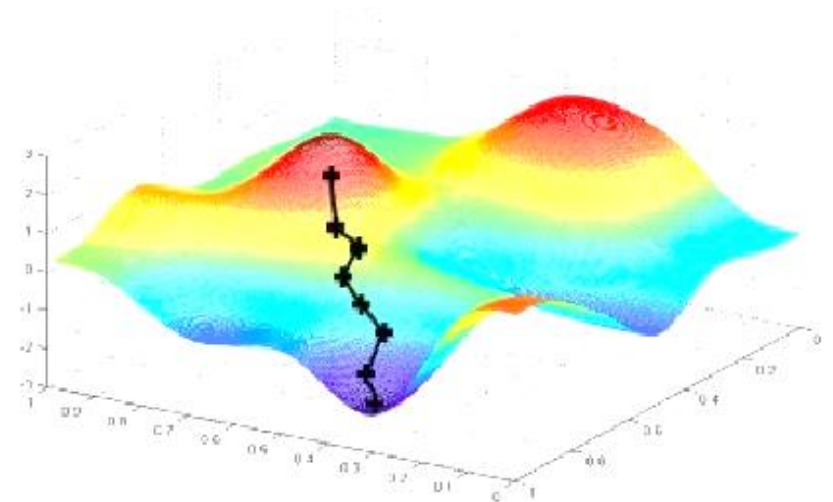


Descenso por Gradiente

Algoritmo

1. Inicializar los pesos aleatoriamente $\sim \mathcal{N}(0, \sigma^2)$
2. Repetir hasta la convergencia:
3. Calcular el gradiente, $\frac{\partial J(W)}{\partial W}$
4. Actualizar los pesos, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Devolver los pesos

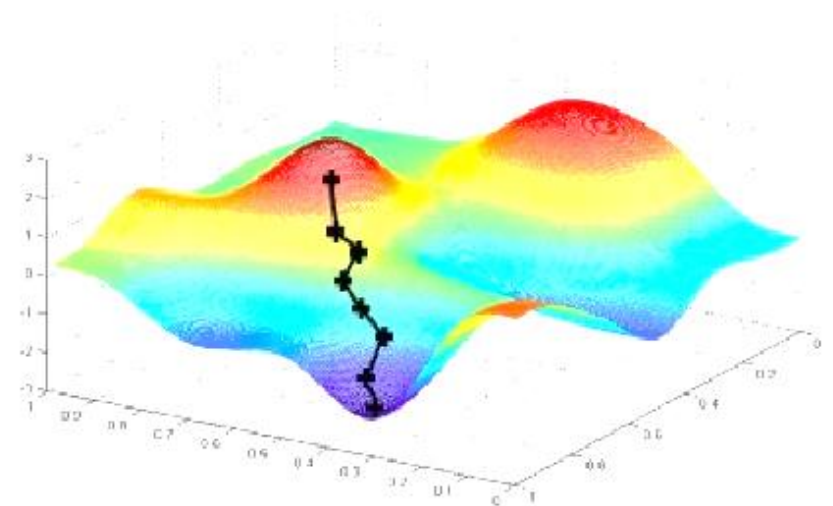
¡Puede ser muy costoso computacionalmente de calcular!



Descenso por Gradiente Estocástico

Algoritmo

1. Inicializar los pesos aleatoriamente $\sim \mathcal{N}(0, \sigma^2)$
2. Repetir hasta la convergencia:
3. Escoger un único punto de datos i
4. Calcular el gradiente, $\frac{\partial J_i(W)}{\partial W}$
5. Actualizar los pesos, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
6. Devolver los pesos

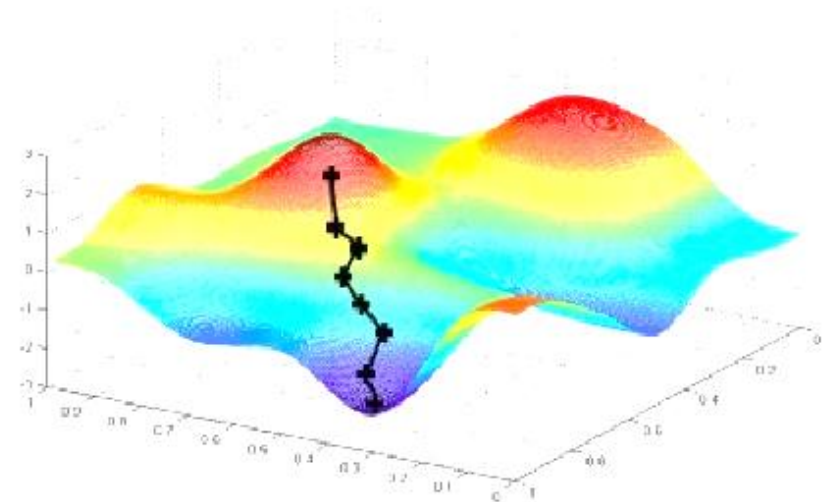


Descenso por Gradiente Estocástico

Algoritmo

1. Inicializar los pesos aleatoriamente $\sim \mathcal{N}(0, \sigma^2)$
2. Repetir hasta la convergencia:
3. Escoger un único punto de datos i
4. Calcular el gradiente, $\frac{\partial J_i(W)}{\partial W}$
5. Actualizar los pesos, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
6. Devolver los pesos

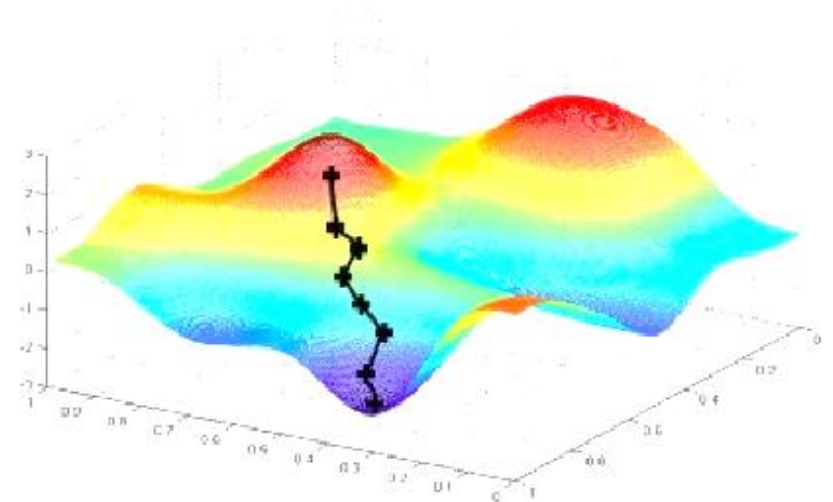
Fácil de calcular pero muy ruidoso (estocástico)



Descenso por Gradiente Estocástico

Algoritmo

1. Inicializar los pesos aleatoriamente $\sim \mathcal{N}(0, \sigma^2)$
2. Repetir hasta la convergencia:
3. Seleccionar un lote de B puntos de datos
4. Calcular el gradiente, $\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(W)}{\partial W}$
5. Actualizar los pesos, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
6. Devolver los pesos

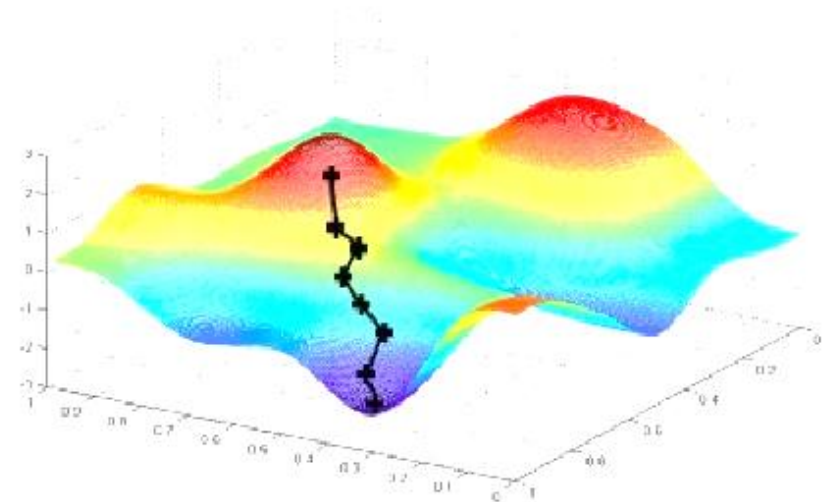


Descenso por Gradiente Estocástico

Algoritmo

1. Inicializar los pesos aleatoriamente $\sim \mathcal{N}(0, \sigma^2)$
2. Repetir hasta la convergencia:
3. Seleccionar un lote de B puntos de datos
4. Calcular el gradiente, $\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(W)}{\partial W}$
5. Actualizar los pesos, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
6. Devolver los pesos

¡Rápido de calcular y una estimación mucho mejor del gradiente verdadero!



Estimación más precisa del gradiente

Convergencia más suave

Permite tasas de aprendizaje más grandes

¡Los mini-lotes conducen a un entrenamiento más rápido!

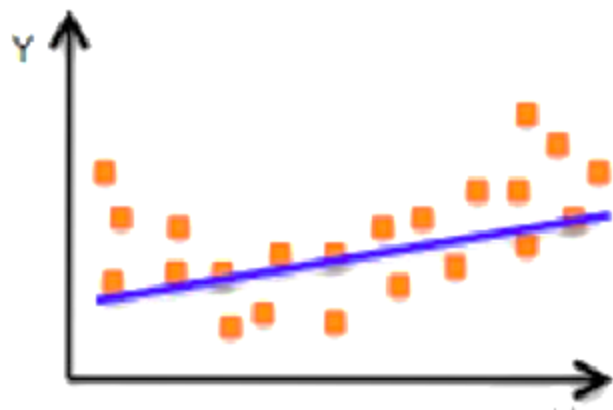
Se puede paralelizar la computación y lograr incrementos significativos de velocidad en las GPU



Redes neuronales en la práctica: Sobreaajuste

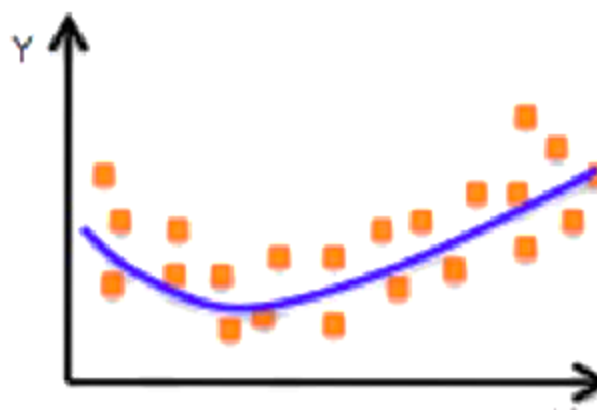


Cómo verificar tus dispositivos CUDA

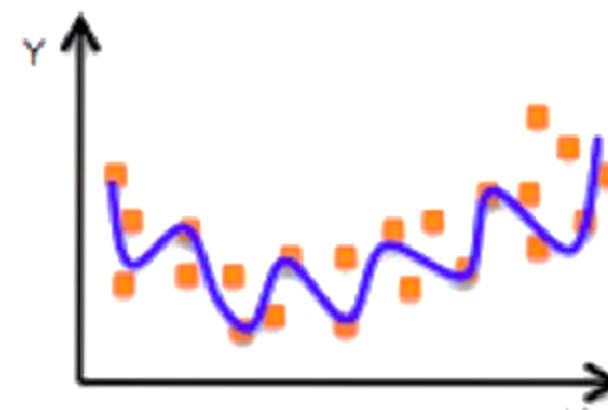


Subajuste

El modelo no tiene la capacidad para aprender completamente los datos



Ajuste ideal



Sobreajuste

Demasiado complejo, con parámetros extra; no generaliza bien



¿Qué es?

Técnica que restringe nuestro problema de optimización para desalentar modelos complejos

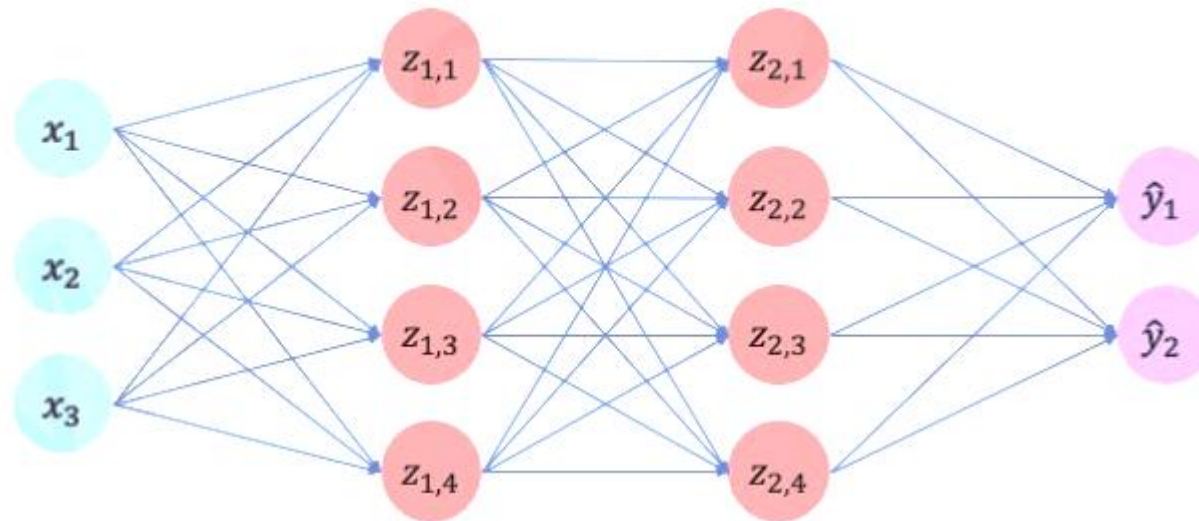
¿Por qué la necesitamos?

Mejorar la generalización de nuestro modelo en datos no vistos



Regularización 1: Dropout

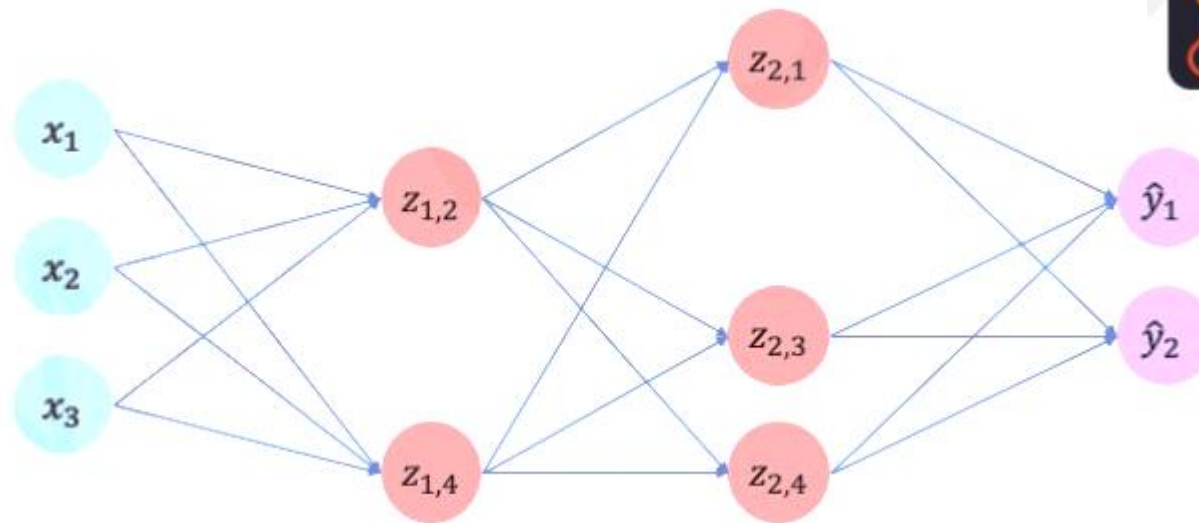
Durante el entrenamiento, poner aleatoriamente algunas activaciones en 0



Regularización 1: Dropout

Durante el entrenamiento, poner aleatoriamente algunas activaciones en 0

- Normalmente se "eliminan" el 50% de las activaciones en la capa
- Obliga a la red a no depender de un solo nodo



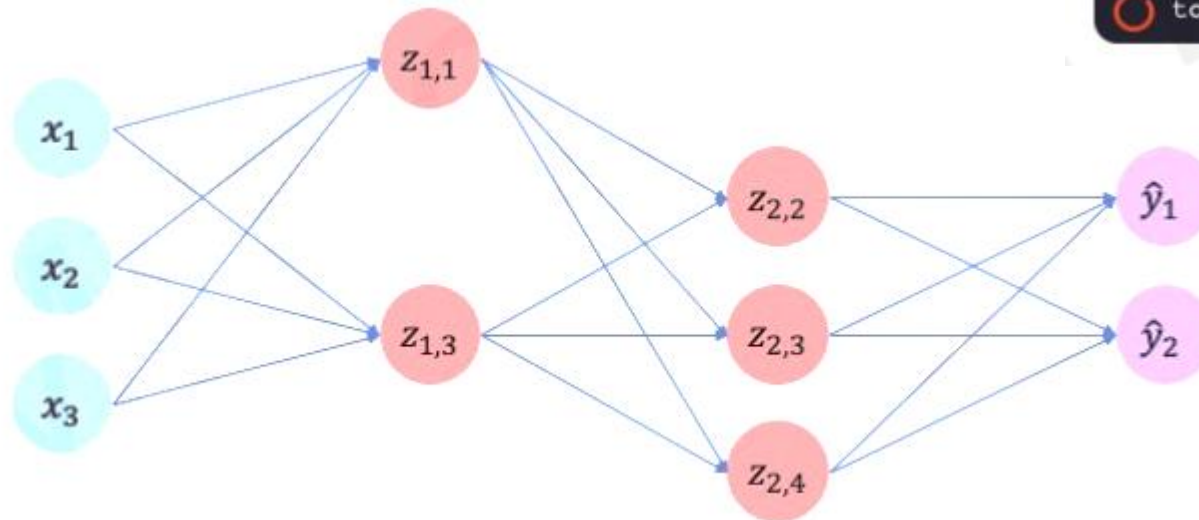
```
tf.keras.layers.Dropout(p=0.5)  
torch.nn.Dropout(p=0.5)
```



Regularización 1: Dropout

Durante el entrenamiento, poner aleatoriamente algunas activaciones en 0

- Normalmente se "eliminan" el 50% de las activaciones en la capa
- Obliga a la red a no depender de un solo nodo

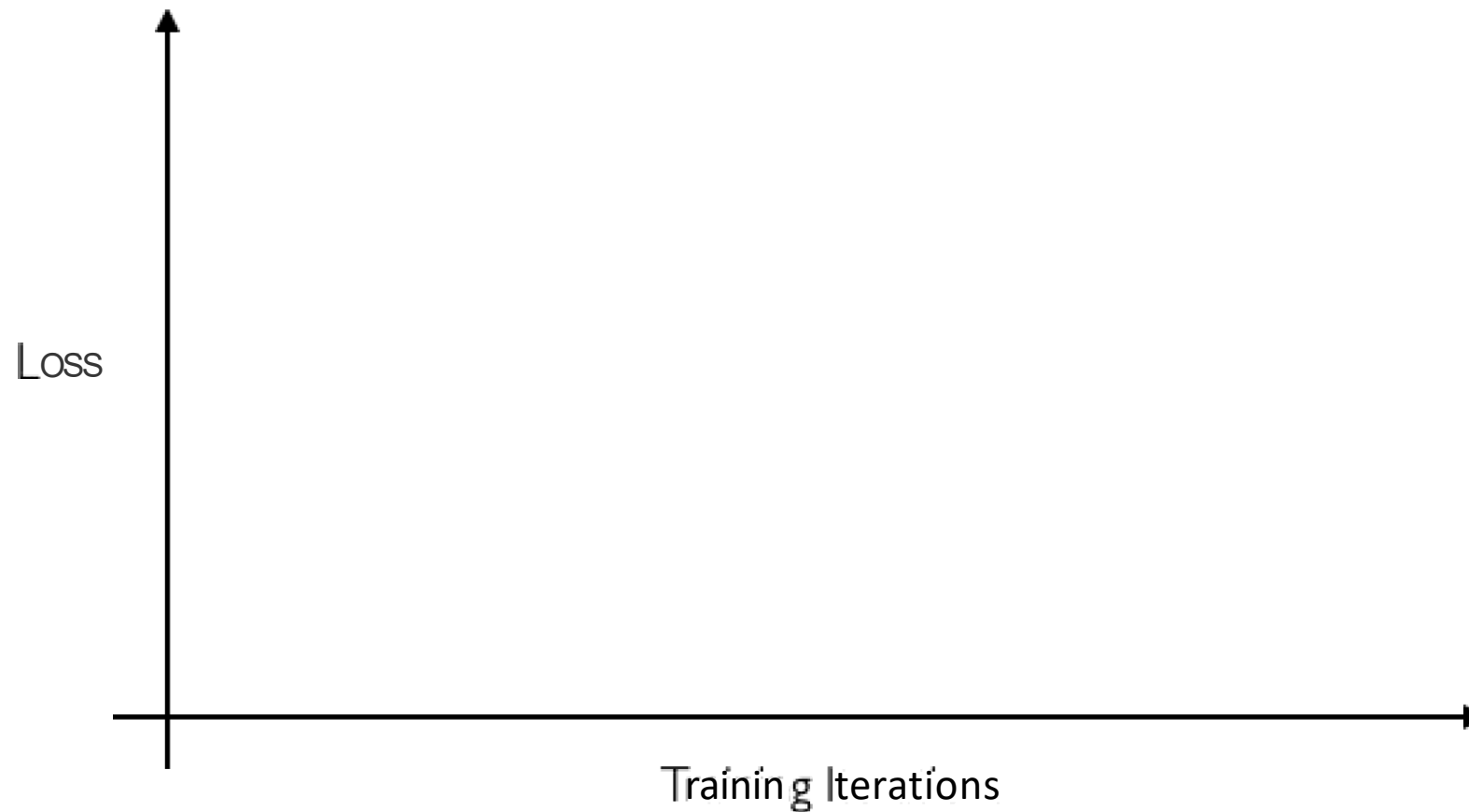


```
tf.keras.layers.Dropout(p=0.5)  
torch.nn.Dropout(p=0.5)
```



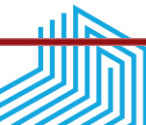
Regularización 2: Detención temprana (Early Stopping)

- Detener el entrenamiento antes de que tengamos la oportunidad de sobreajustar



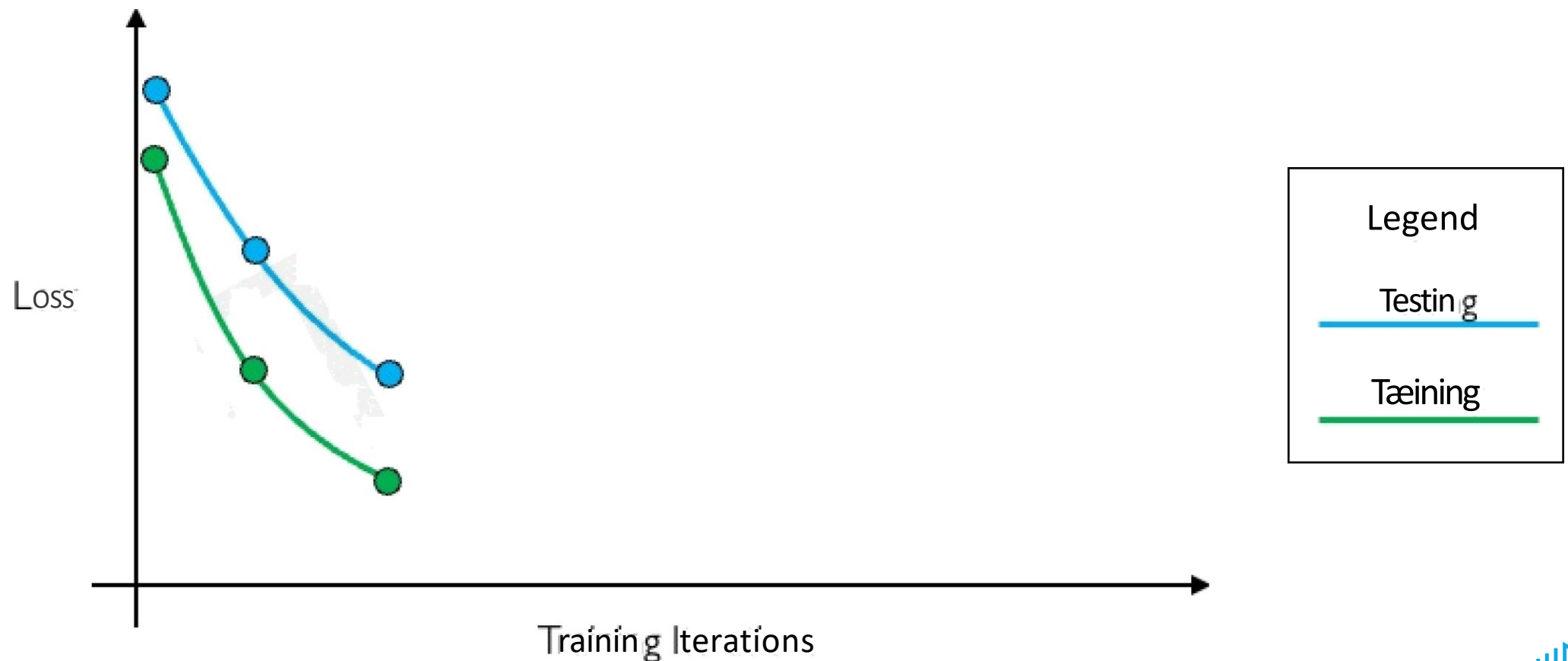
Regularización 2: Detención temprana (Early Stopping)

- Detener el entrenamiento antes de que tengamos la oportunidad de sobreajustar



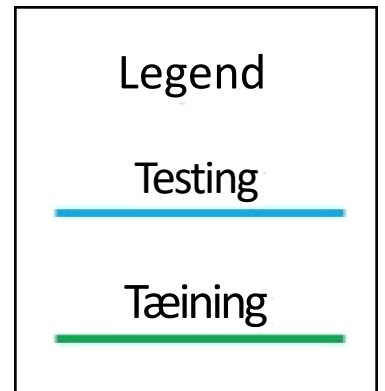
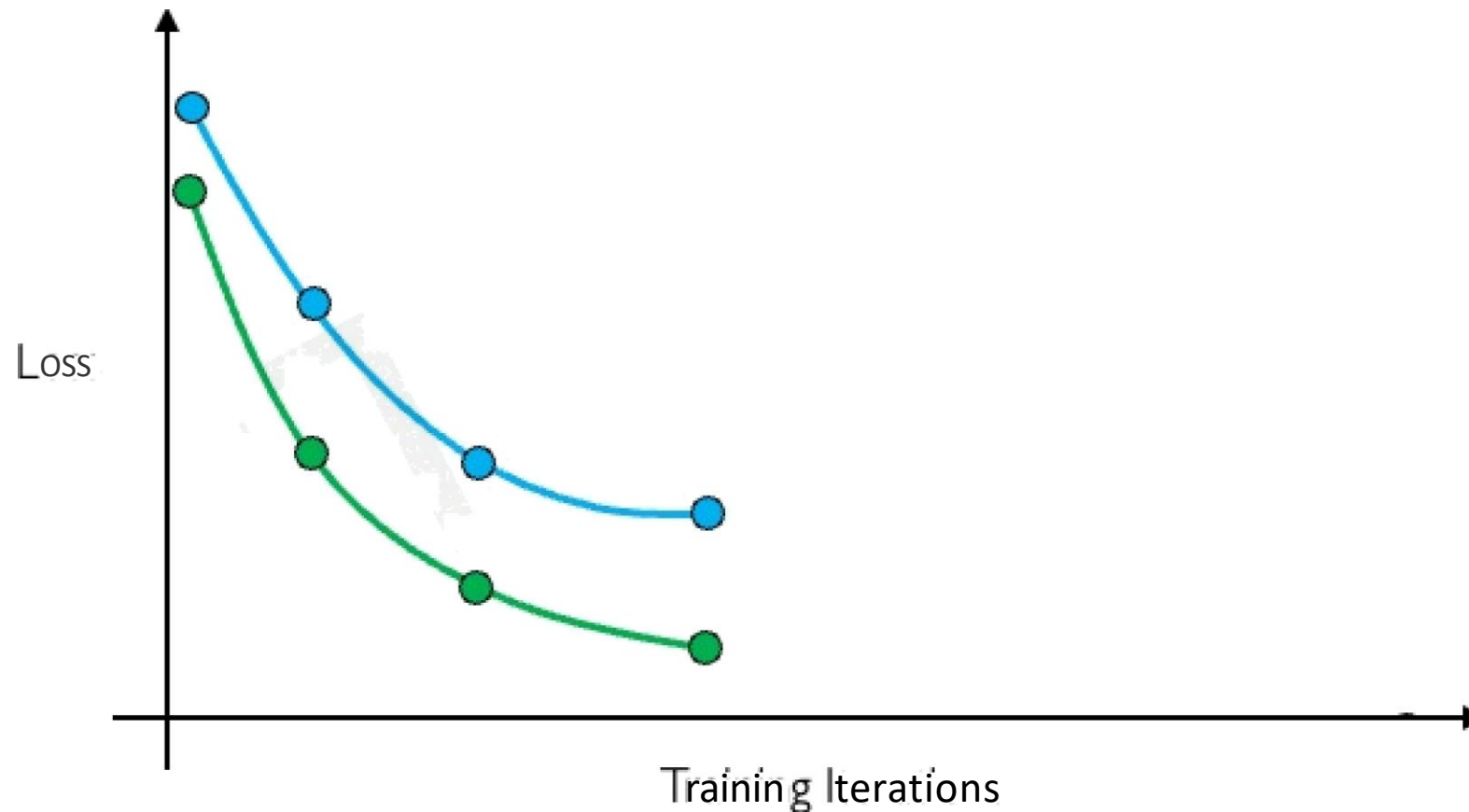
Regularización 2: Detención temprana (Early Stopping)

- Detener el entrenamiento antes de que tengamos la oportunidad de sobreajustar



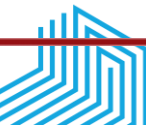
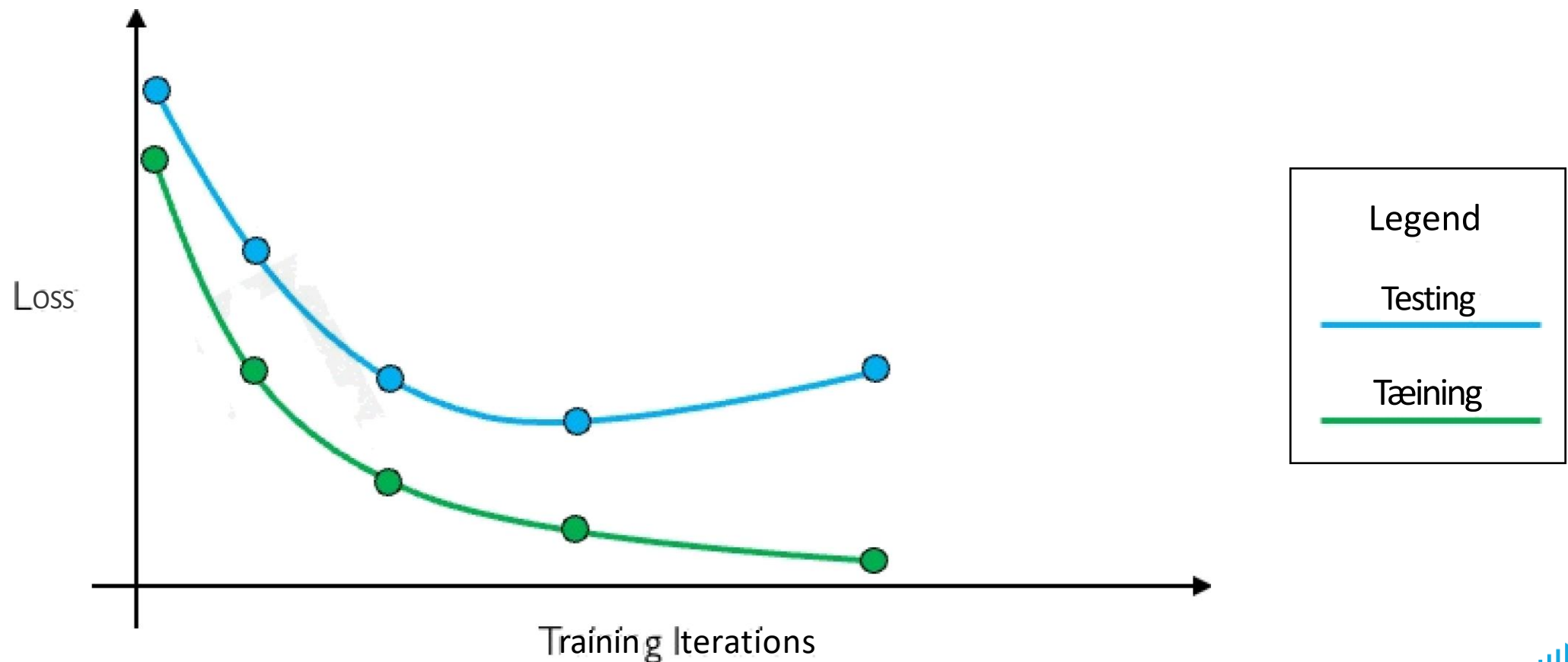
Regularización 2: Detención temprana (Early Stopping)

- Detener el entrenamiento antes de que tengamos la oportunidad de sobreajustar



Regularización 2: Detención temprana (Early Stopping)

- Detener el entrenamiento antes de que tengamos la oportunidad de sobreajustar



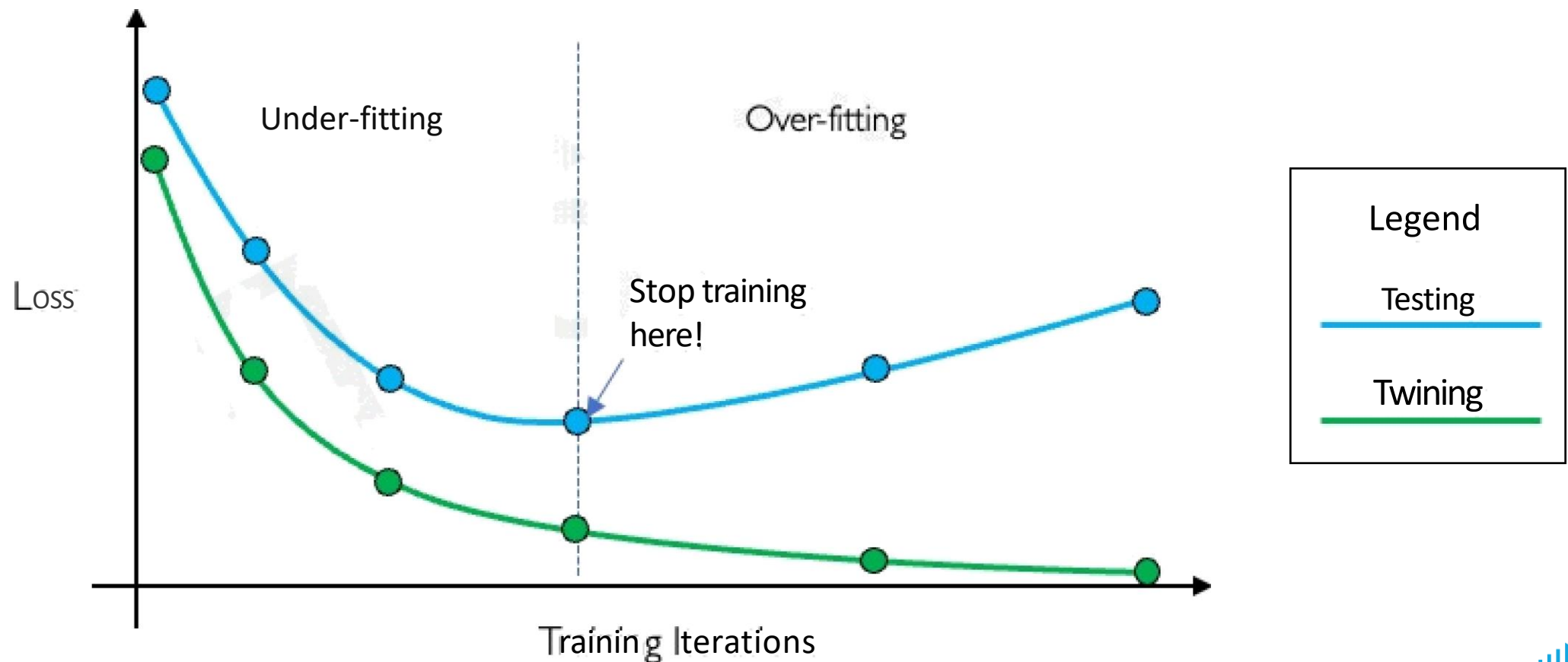
Regularización 2: Detención temprana (Early Stopping)

- Detener el entrenamiento antes de que tengamos la oportunidad de sobreajustar



Regularización 2: Detención temprana (Early Stopping)

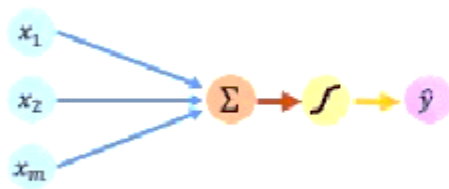
- Detener el entrenamiento antes de que tengamos la oportunidad de sobreajustar



Revisión de Fundamentos Básicos

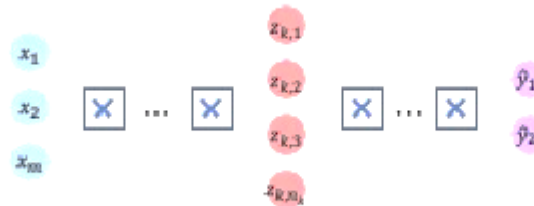
El perceptrón

- Bloques estructurales de construcción
- Funciones de activación no lineales



Redes neuronales

- Apilamiento de perceptrones para formar redes neuronales
- Optimización mediante retropropagación



Entrenamiento en la práctica

- Aprendizaje adaptativo
- Agrupamiento en lotes (batching)
- Regularización

