

Deep Learning

Introducción y Redes Neuronales



Marco Teran



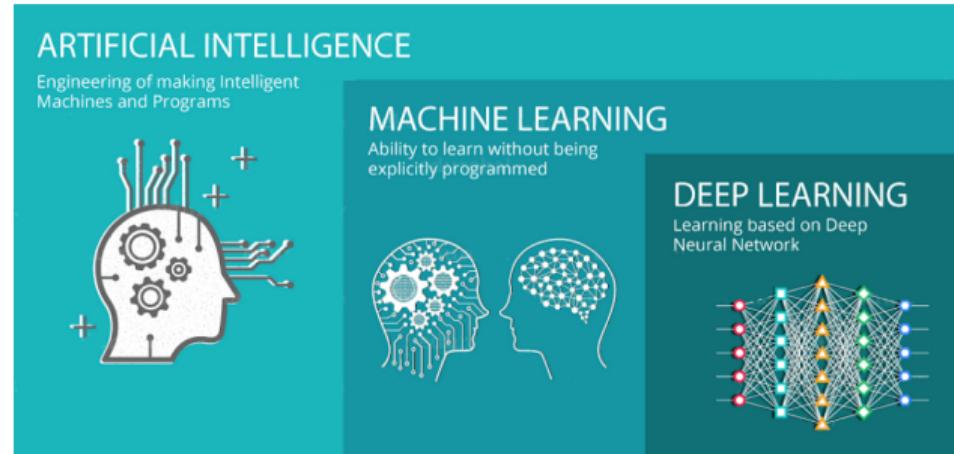
UNIVERSIDAD
SERGIO ARBOLEDA

Octubre 2020- Bogotá

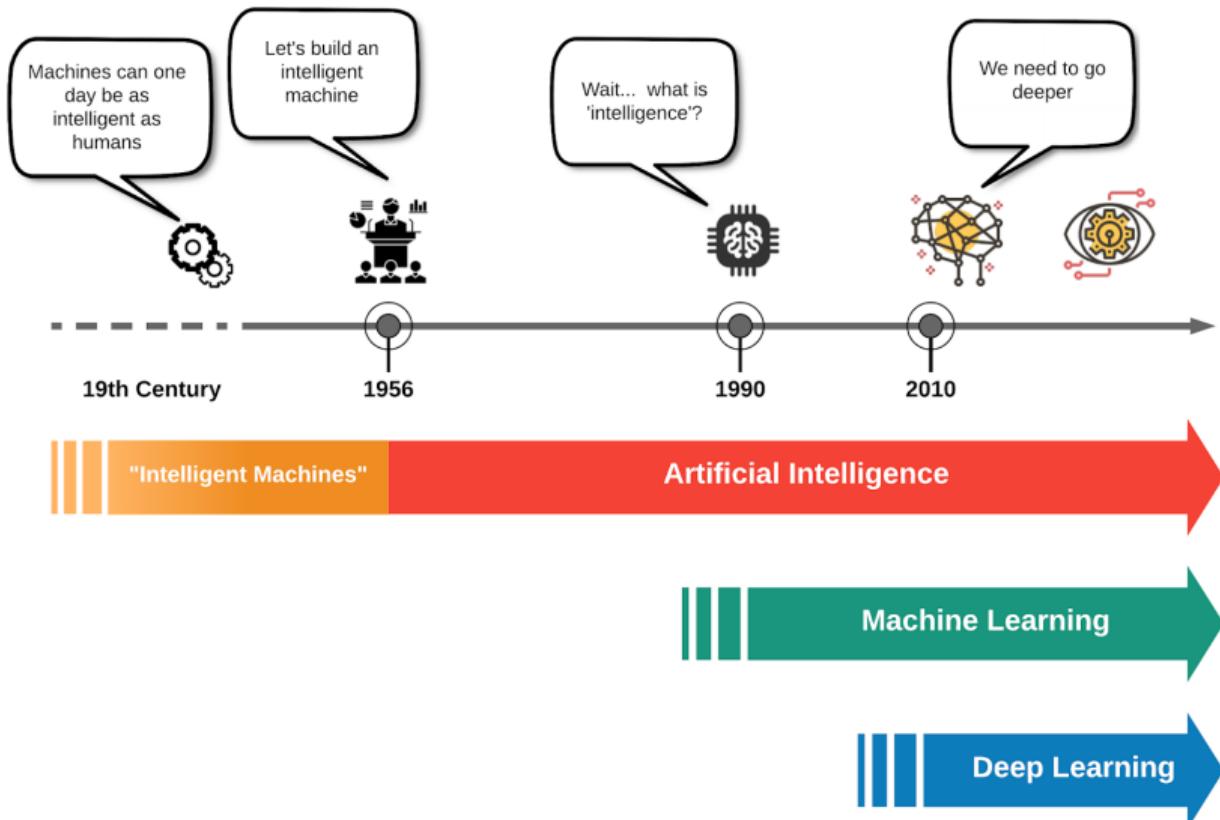
Outline

- 1 Introducción
 - 2 El Perceptrón
 - Funciones de activación
 - El perceptrón: ejemplo
 - 3 Construyendo redes neuronales con Perceptrones
 - 4 Aplicando redes neuronales
 - 5 Entrenando redes neuronales
 - 6 Redes neuronales en la práctica: Optimización
 - 7 Redes neuronales en la práctica: Mini-lotes
 - 8 Redes neuronales en la práctica: Sobreajuste (overfitting)

¿Qué es el Aprendizaje profundo (Deep Learning)?



- **Inteligencia Artificial (Artificial Intelligence)**: Cualquier técnica que permita a los ordenadores imitar el comportamiento humano
 - **Aprendizaje de maquina (Machine Learning)**: Capacidad de aprender sin ser programado explícitamente
 - **Aprendizaje profundo (Deep Learning)**: Extraer patrones de datos utilizando redes neuronales (neural networks)



Machine Learning



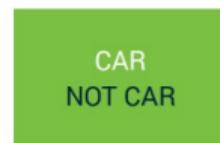
Input



Feature extraction



Classification



Output

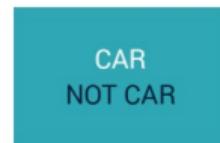
Deep Learning



Input



Feature extraction + Classification

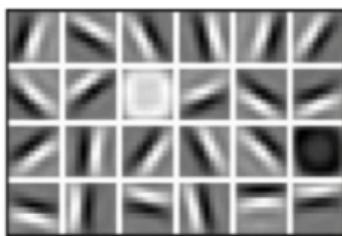


Output

¿Por qué Deep Learning y por qué ahora?

¿Por qué el Deep Learning?

- Las características diseñadas a mano consumen mucho tiempo, son frágiles y no se pueden escalar en la práctica.
- ¿Podemos aprender las **características subyacentes** directamente de los datos?



(a) Características de bajo nivel: Líneas y bordes

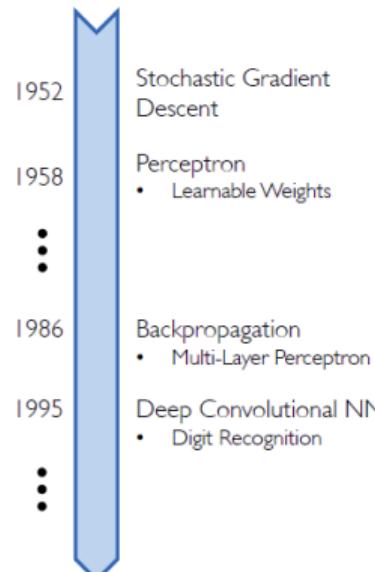


(b) Características de nivel medio: Ojos, nariz y oídos



(c) Características de alto nivel: Estructura facial

¿Por qué ahora?



Las redes neuronales se remontan a décadas atrás, así que ¿por qué el resurgimiento?

1. Big Data

- Conjuntos de datos más grandes
- Recolección y almacenamiento más fácil

IMAGENET



2. Hardware

- Unidades de procesamiento gráfico (GPU)
- Masivamente paralelizable



3. Software

- Técnicas mejoradas
- Nuevos modelos
- Toolboxes

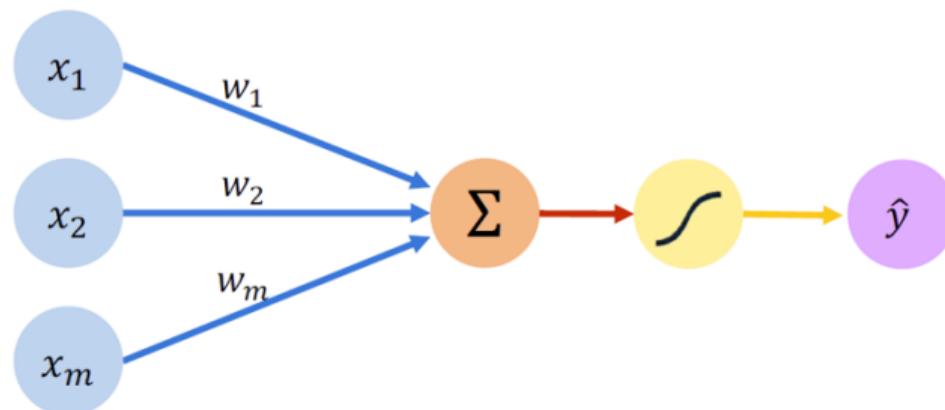


El Perceptrón

La base estructural del aprendizaje profundo

El Perceptrón: Propagación hacia adelante

Forward Propagation



Combinación lineal de las entradas

Salida

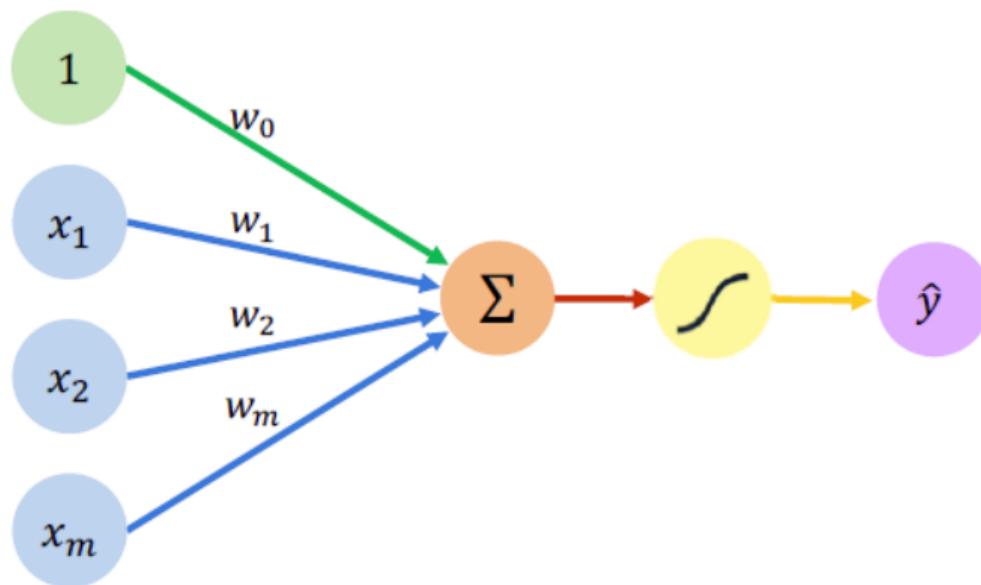
$$\hat{y} = g \left(\sum_{i=1}^m x_i w_i \right)$$

Función de activación no lineal

Entrada Pesos Suma No linealidad Salida

El Perceptrón: Propagación hacia adelante

Forward Propagation



Combinación lineal de las entradas

Salida

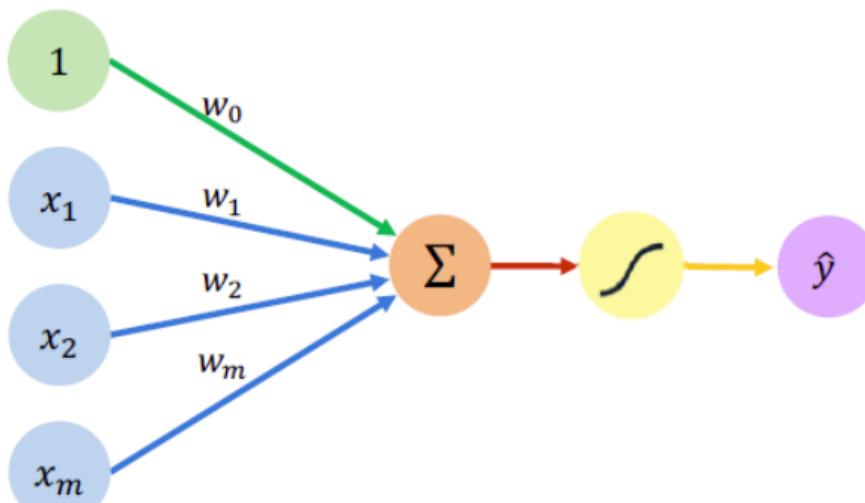
$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

Bias

Función de activación no lineal

El Perceptrón: Propagación hacia adelante

Forward Propagation



Entradas Pesos

Suma No linealidad Salida

y

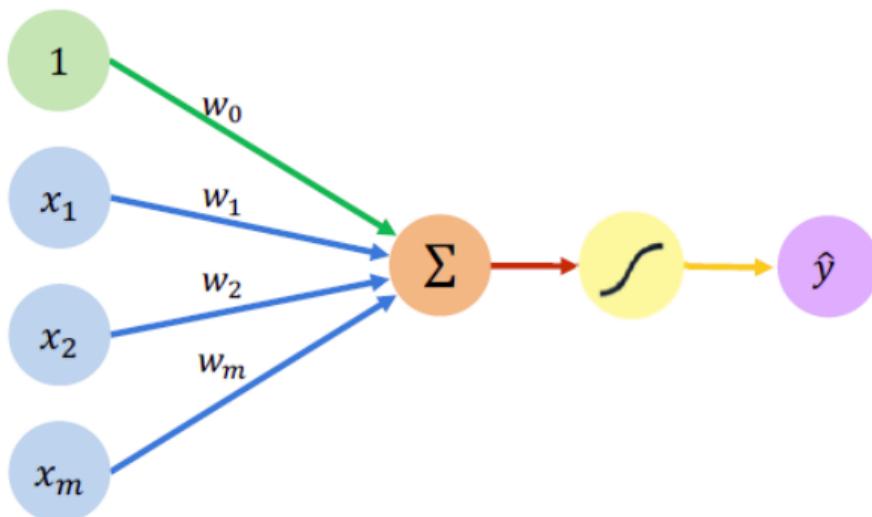
$$\hat{y} = g(w_o + \sum_{i=1}^m x_i w_i)$$

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

donde,

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \quad (1)$$

El Perceptrón: Propagación hacia adelante

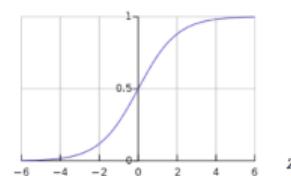


Función de activación

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

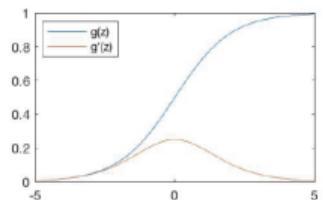
Ejemplo: función sigmoidal

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



Funciones de activación comunes

Función sigmoidal

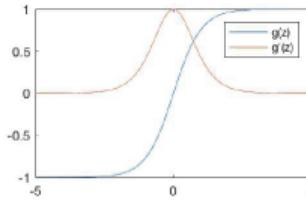


$$g(z) = \frac{1}{1 - e^{-z}}$$

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

```
tf.nn.sigmoid(z)
```

Tangente hiperbólica

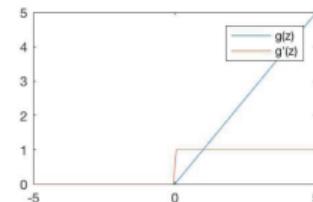


$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

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

`tf.nn.tanh(z)`

Rectified Linear Unit (ReLU)



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

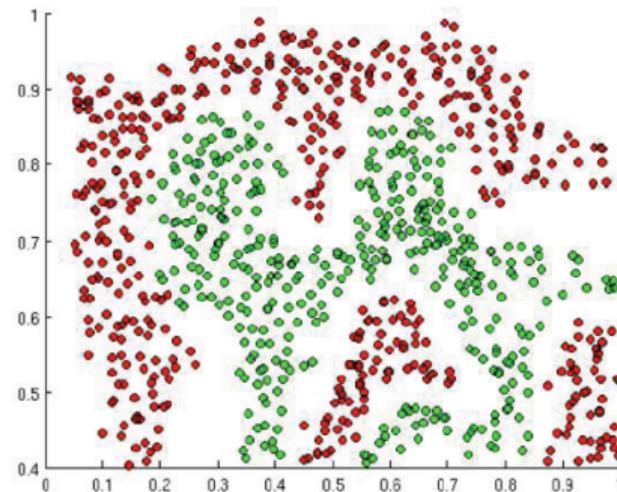
$$g'(z) = \begin{cases} 1, & \text{si } z > 0; \\ 0, & \text{otroscasos} \end{cases}$$

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

NOTA: Todas las funciones de activación son no lineales

La importancia de las Funciones de activación

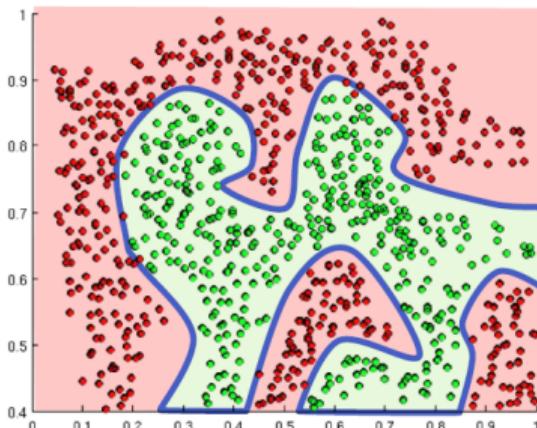
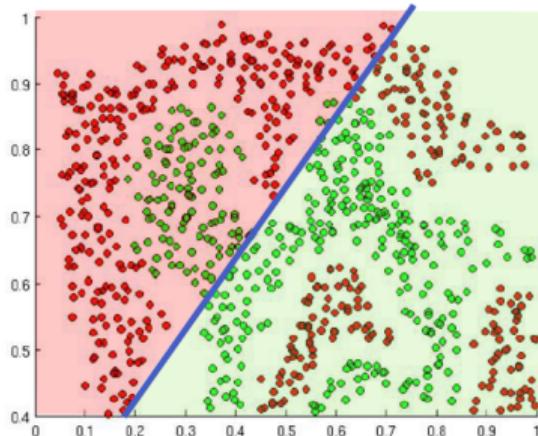
El propósito de las funciones de activación es **introducir las no linealidades** en la red.



¿Y si quisieramos construir una red neuronal para distinguir los puntos verdes de los rojos?

La importancia de las Funciones de activación

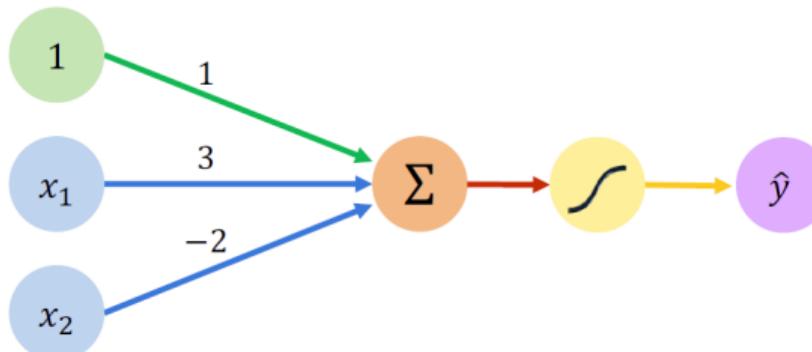
El propósito de las funciones de activación es **introducir las no linealidades** en la red.



Las funciones de Activación Lineal producen decisiones lineales sin importar el tamaño de la red

Las no linealidades nos permiten aproximarlos a funciones arbitrariamente complejas

El perceptrón: ejemplo



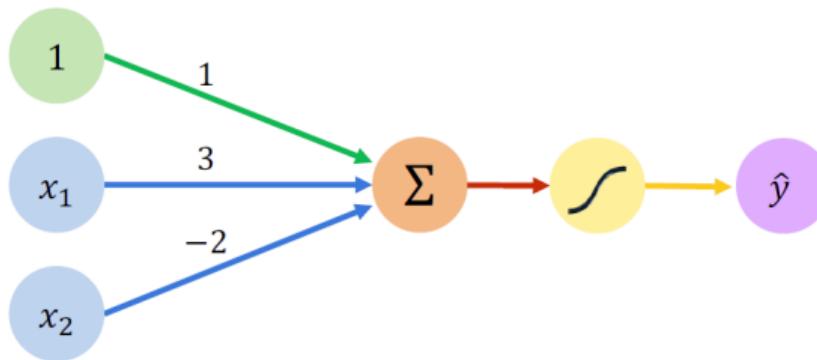
Tenemos: $w_0 = 1$ y $\mathbf{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

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

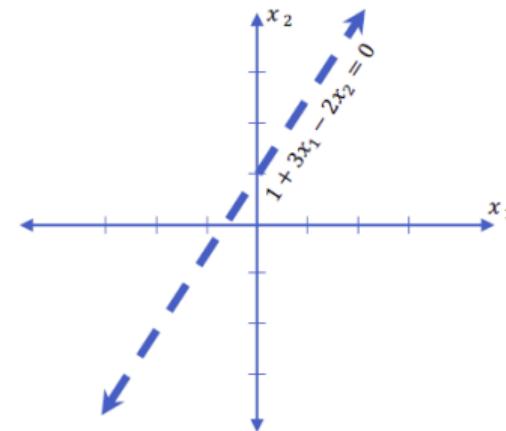
Esto es sólo una línea en 2D

El perceptrón: ejemplo

El perceptrón: ejemplo

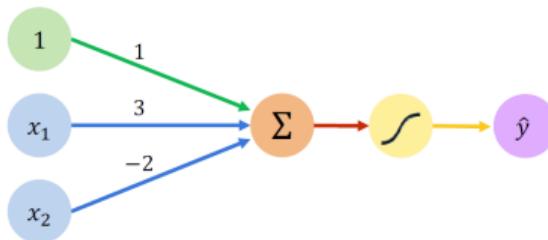


$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

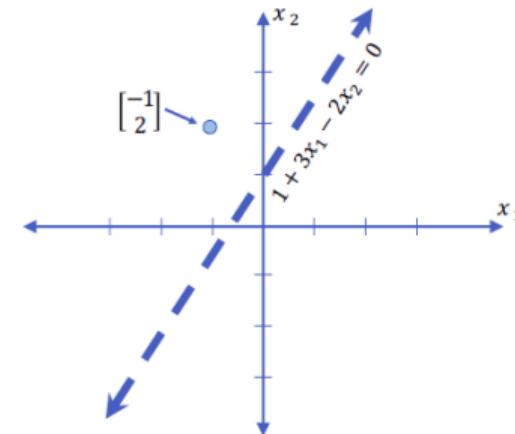


El perceptrón: ejemplo

El perceptrón: ejemplo



$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

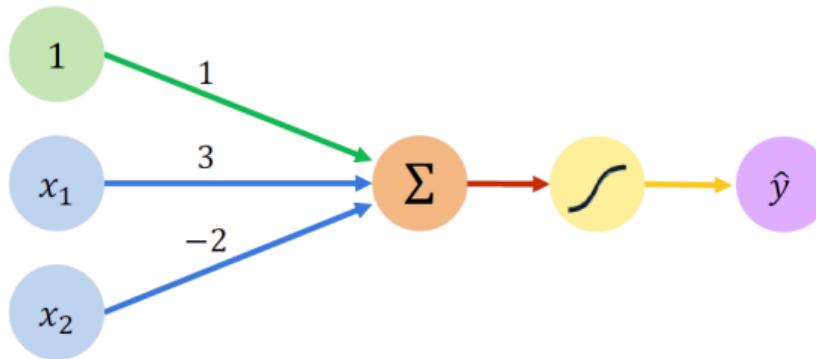


Asumimos una entrada $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

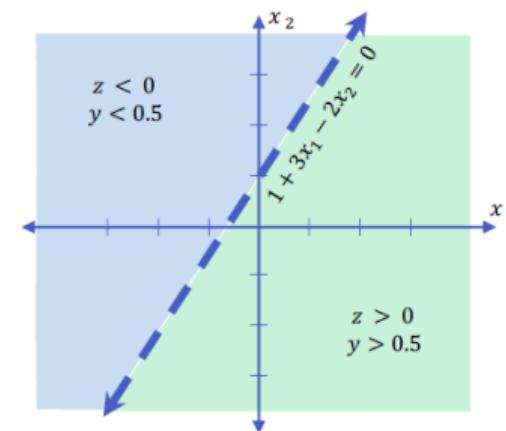
$$\begin{aligned}\hat{y} &= g(1 + 3(-1) - 2(2)) \\ &= g(-6) \approx 0.002\end{aligned}$$

El perceptrón: ejemplo

El perceptrón: ejemplo

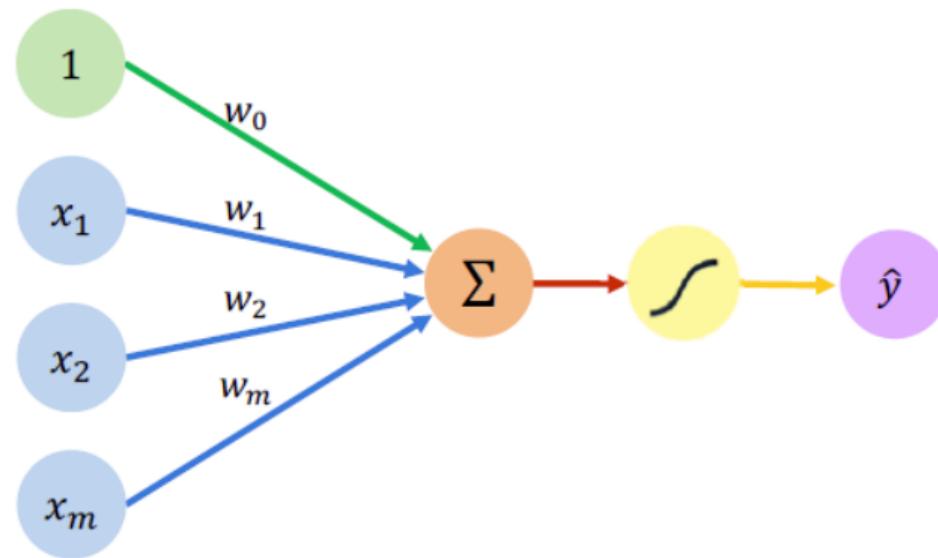


$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



Construyendo redes neuronales con Perceptrones

El perceptrón simplificado



Entradas

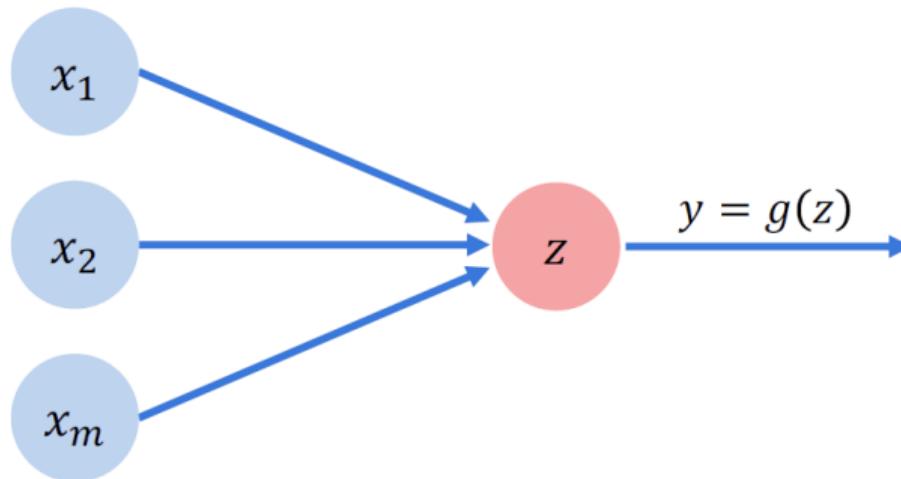
Pesos

Suma

No linealidad

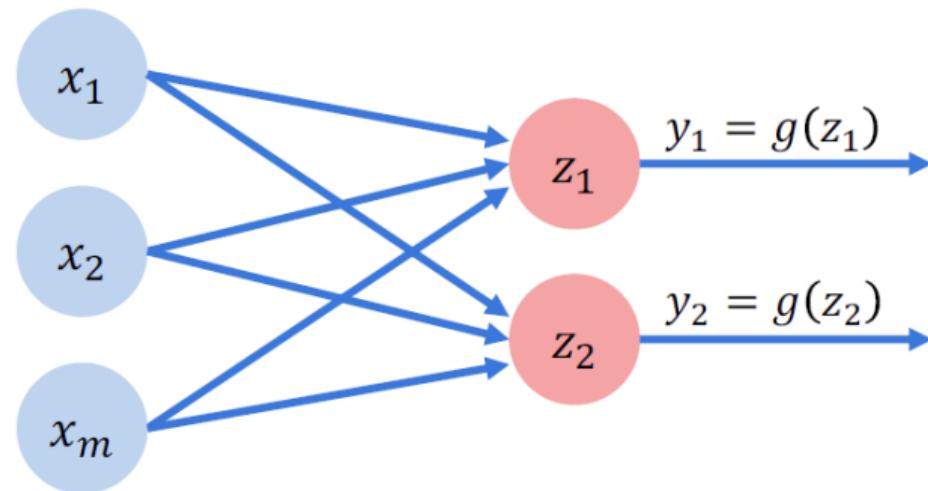
Salida

El perceptrón simplificado



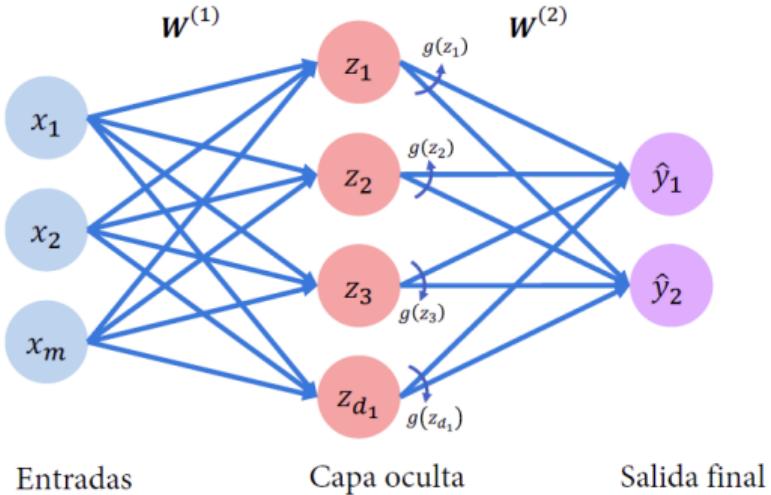
$$z = w_0 + \sum_{j=1}^m x_j w_j$$

Perceptrón de salida múltiple



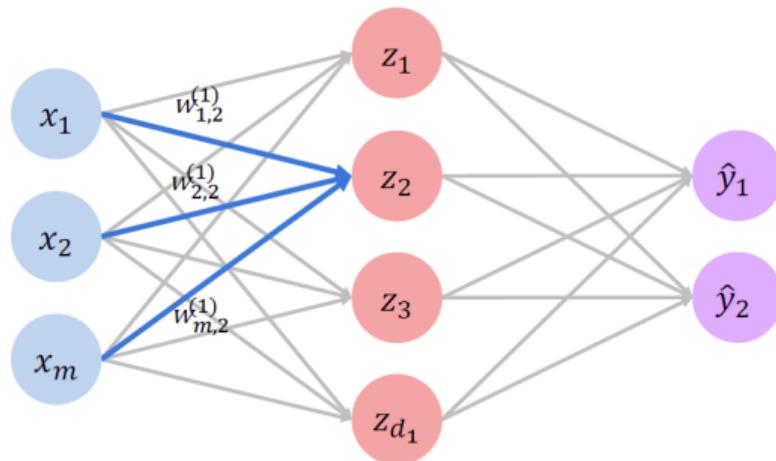
$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$

Red neuronal de una sola capa



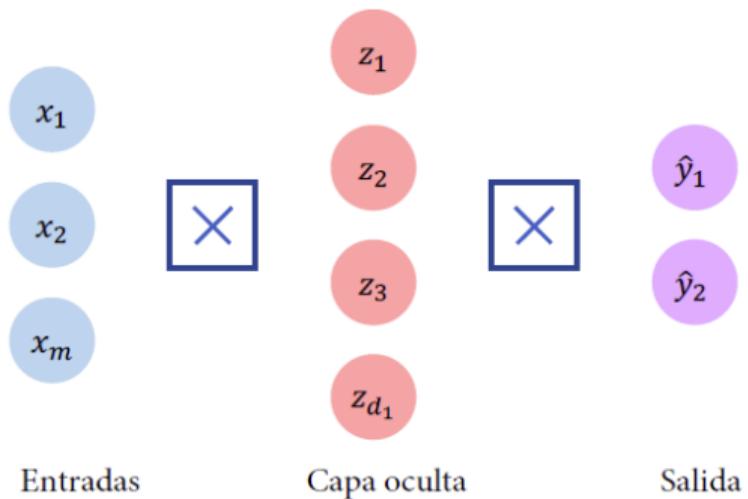
$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^m x_j w_{j,i}^{(1)} \hat{y} = g \left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) w_{j,i}^{(2)} \right)$$

Red neuronal de una sola capa



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

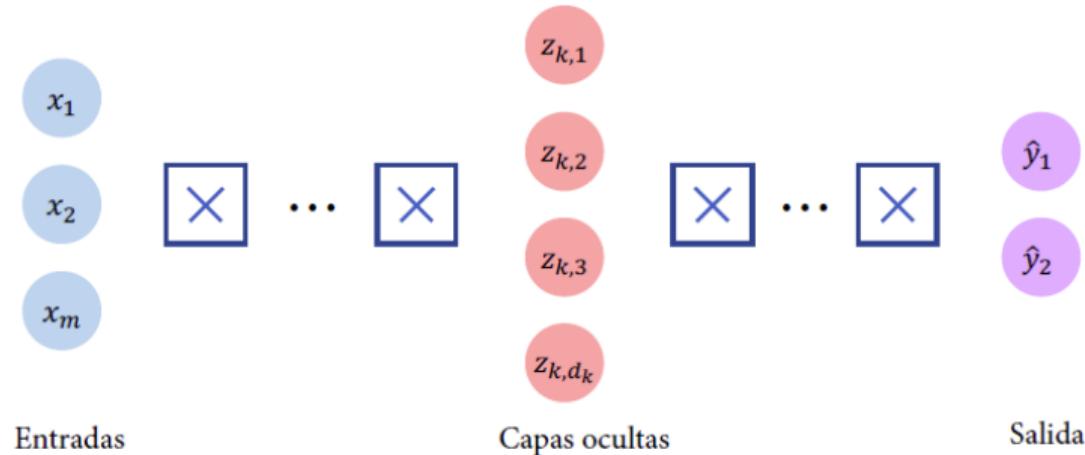
Perceptrón de salida múltiple



```
 from tf.keras.layers import *\n\n    inputs = Inputs(m)\n    hidden = Dense(d1)(inputs)\n    outputs = Dense(2)(hidden)\n    model = Model(inputs, outputs)
```

Red Neuronal profunda

Deep Neural Network



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

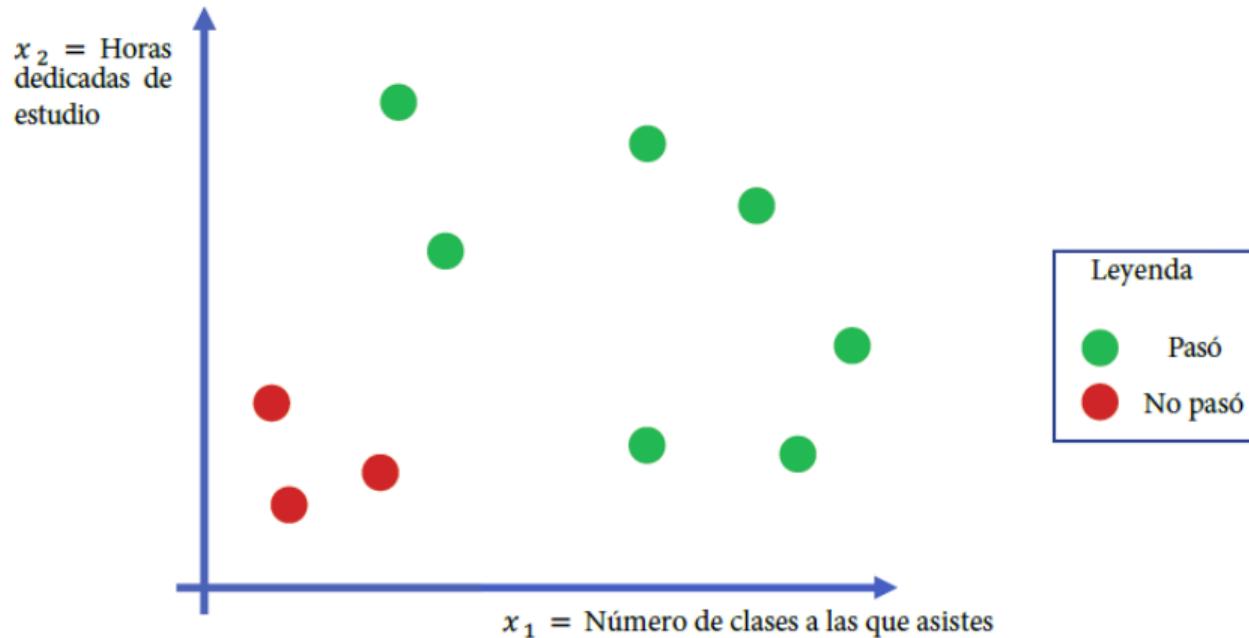
Aplicando redes neuronales

Problema de ejemplo

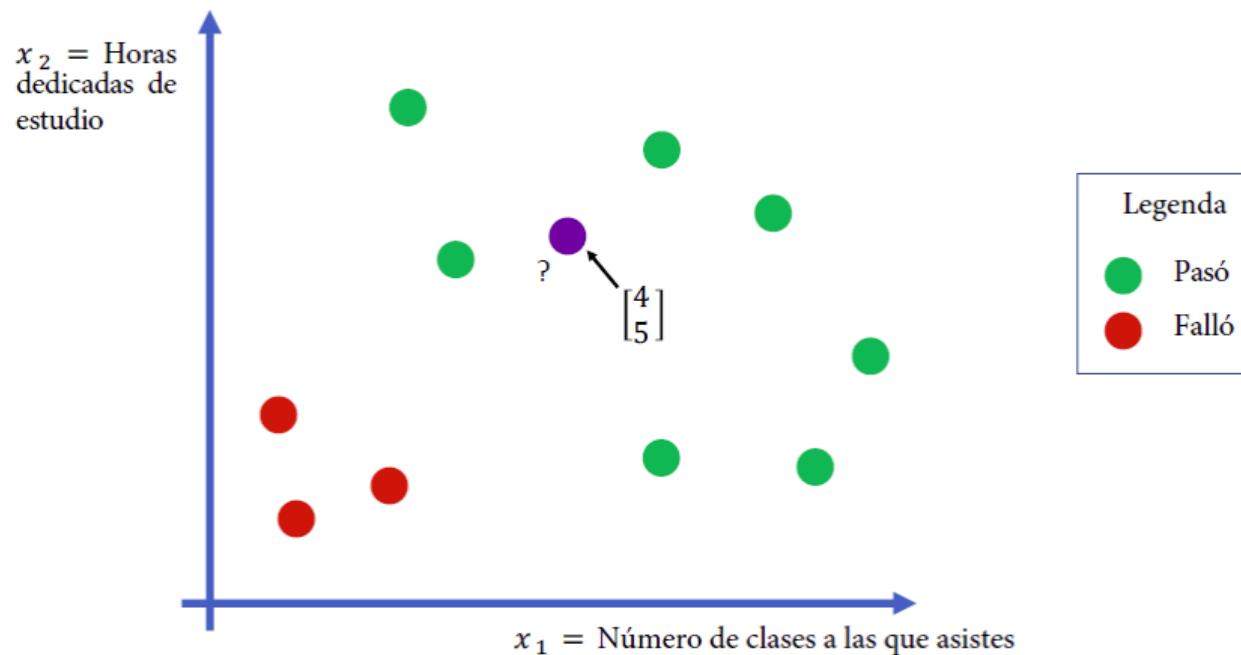
¿Pasaré esta clase?

Empecemos con un simple modelo de dos características:
 x_1 – Número de clases a las que asistes x_2 – Horas dedicadas de estudio

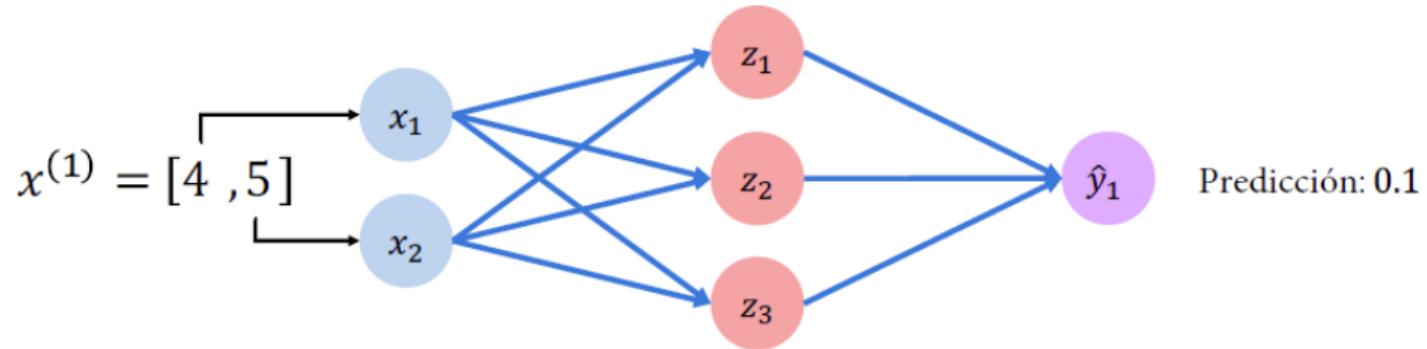
Problema de ejemplo: ¿Pasaré esta clase?



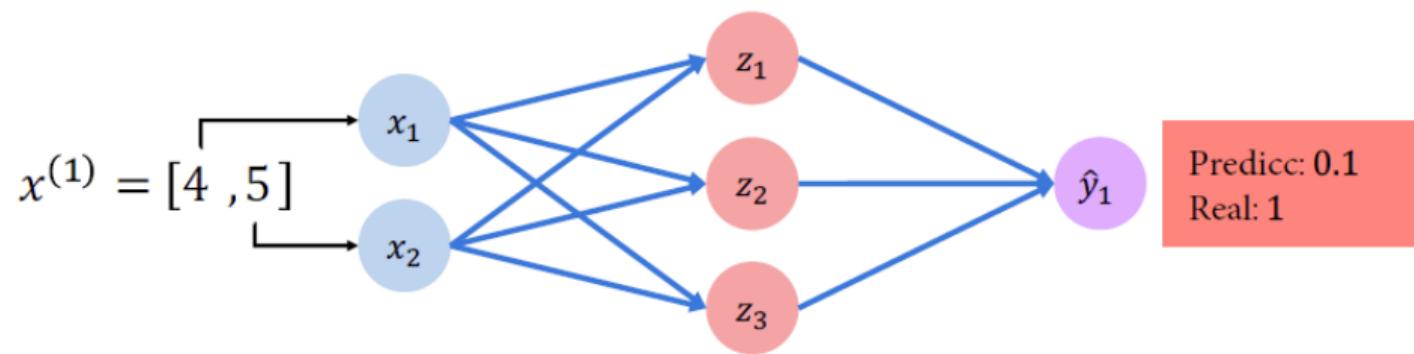
Problema de ejemplo: ¿Pasaré esta clase?



Problema de ejemplo: ¿Pasaré esta clase?

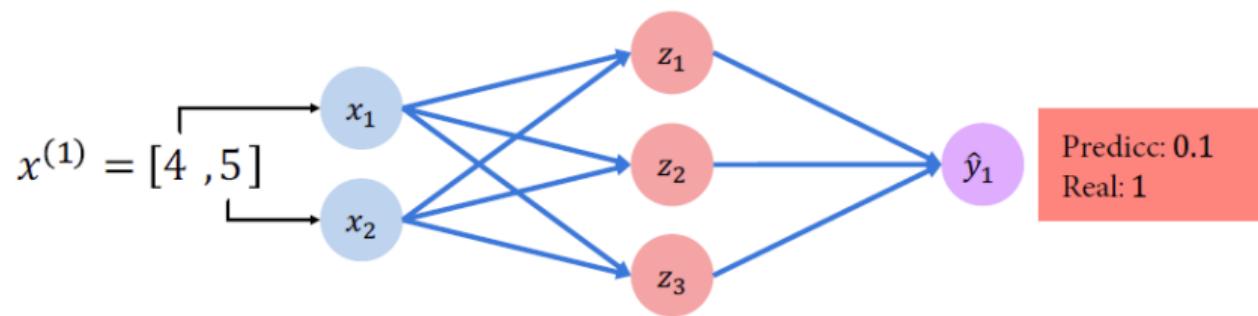


Problema de ejemplo: ¿Pasaré esta clase?



Cuantificación de las pérdidas

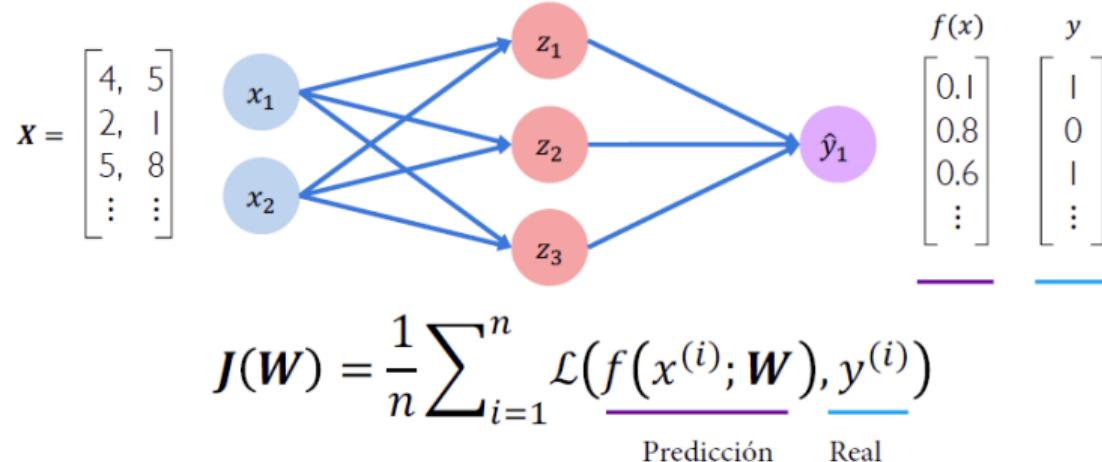
La pérdida (loss) de nuestra red mide el costo incurrido por las predicciones incorrectas



$$\frac{\mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})}{\text{Predicción}} \quad \frac{}{\text{Real}}$$

Pérdidas empíricas

La pérdida empírica mide la pérdida total en todo nuestro conjunto de datos

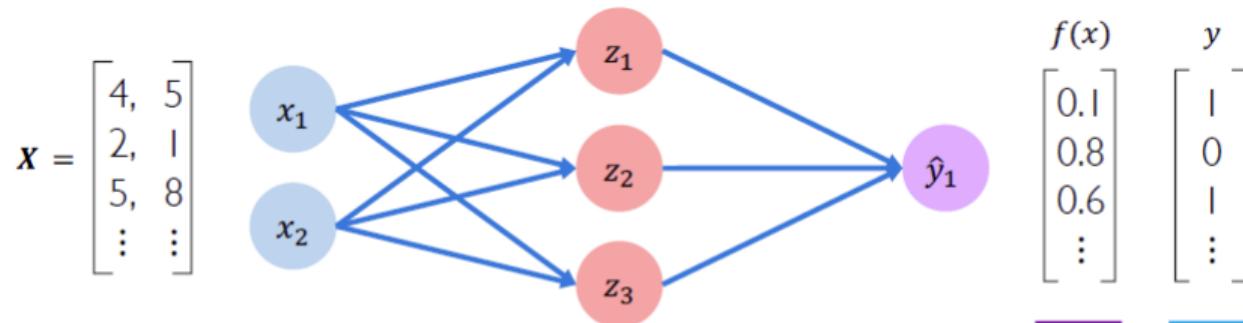


También conocida:

- Función objetivo
- Función de costo
- Riesgo empírico

Entropía cruzada binaria

Binary Cross Entropy Loss La pérdida de entropía cruzada puede utilizarse con modelos que arrojan una probabilidad entre 0 y 1



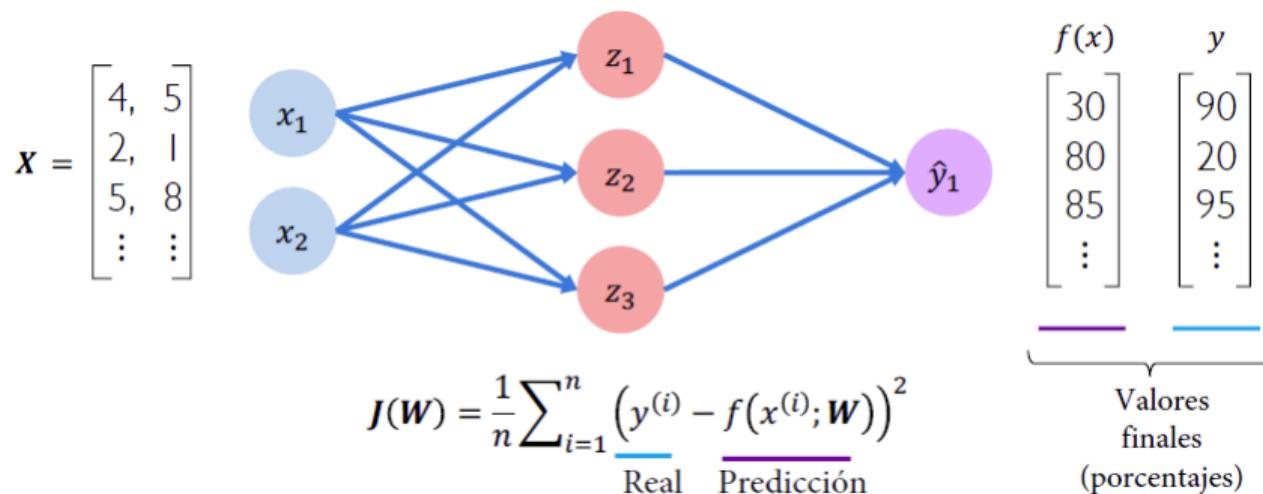
$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)} \log(f(x^{(i)}; \mathbf{W}))}_{\text{Real}} + \underbrace{(1 - y^{(i)}) \log(1 - f(x^{(i)}; \mathbf{W}))}_{\text{Real}} \quad \underbrace{\text{Predicción}}_{\text{Predicción}}$$



```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )
```

Perdida del Error cuadrático medio

Mean Squared Error Loss La pérdida media de error al cuadrado se puede usar con modelos de regresión que producen números reales continuos



```
loss = tf.reduce_mean( tf.square(tf.subtract(model.y, model.pred)) )
```

Entrenando redes neuronales

Optimización de la perdida

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

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

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

Optimización de la perdida

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

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

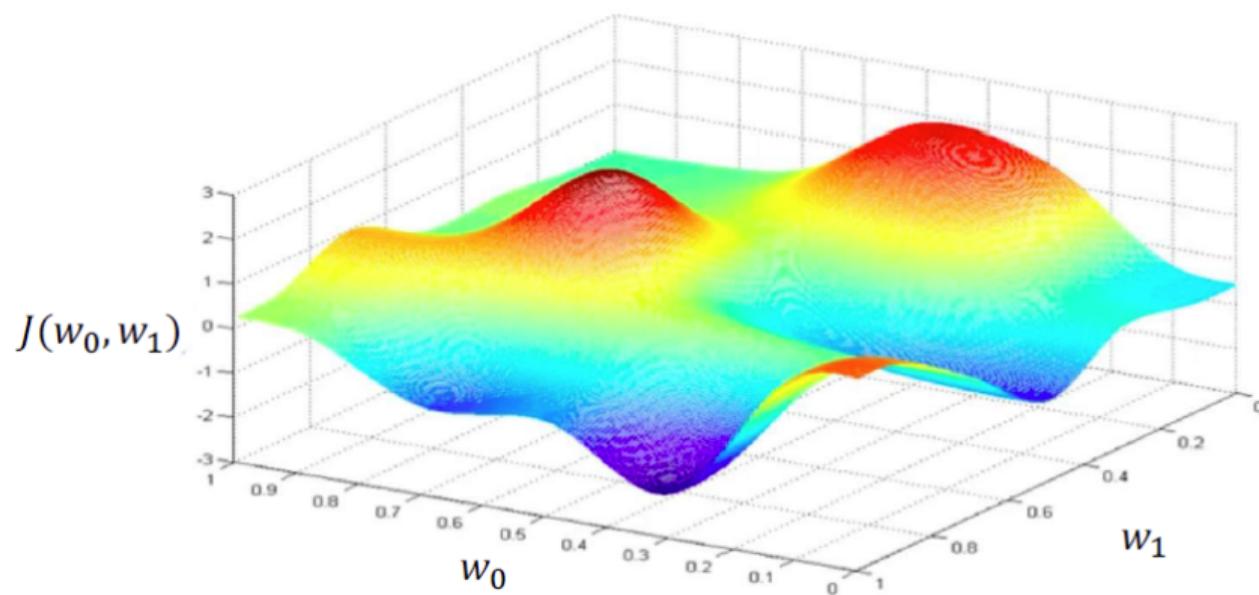
$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$

Recuerda:

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

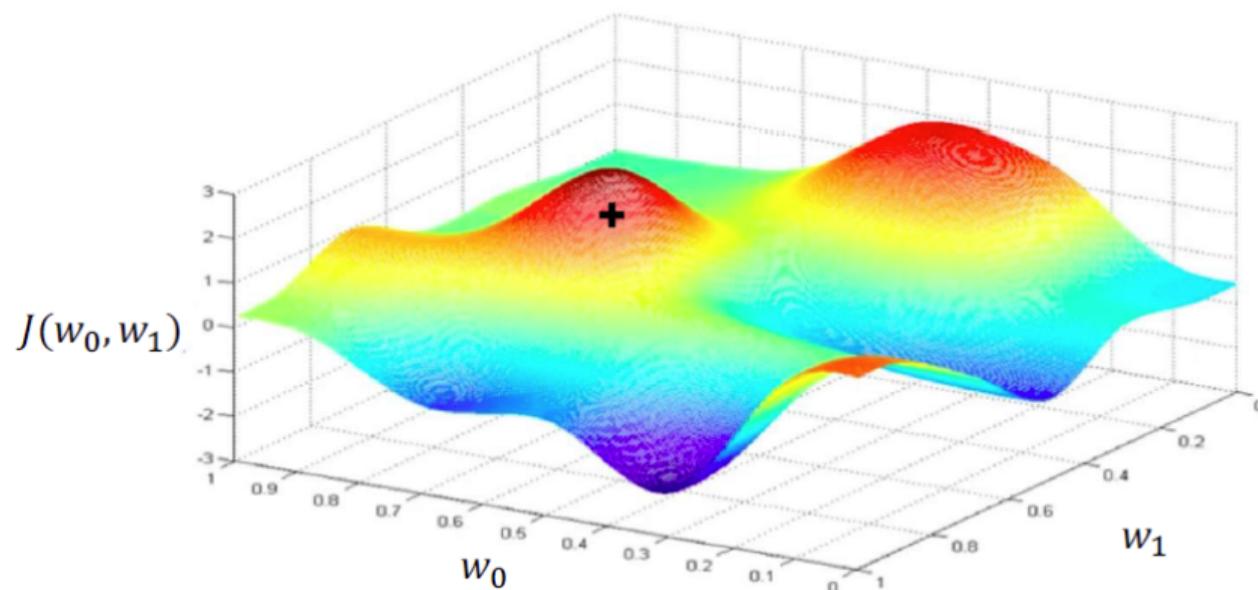
Optimización de la perdida

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



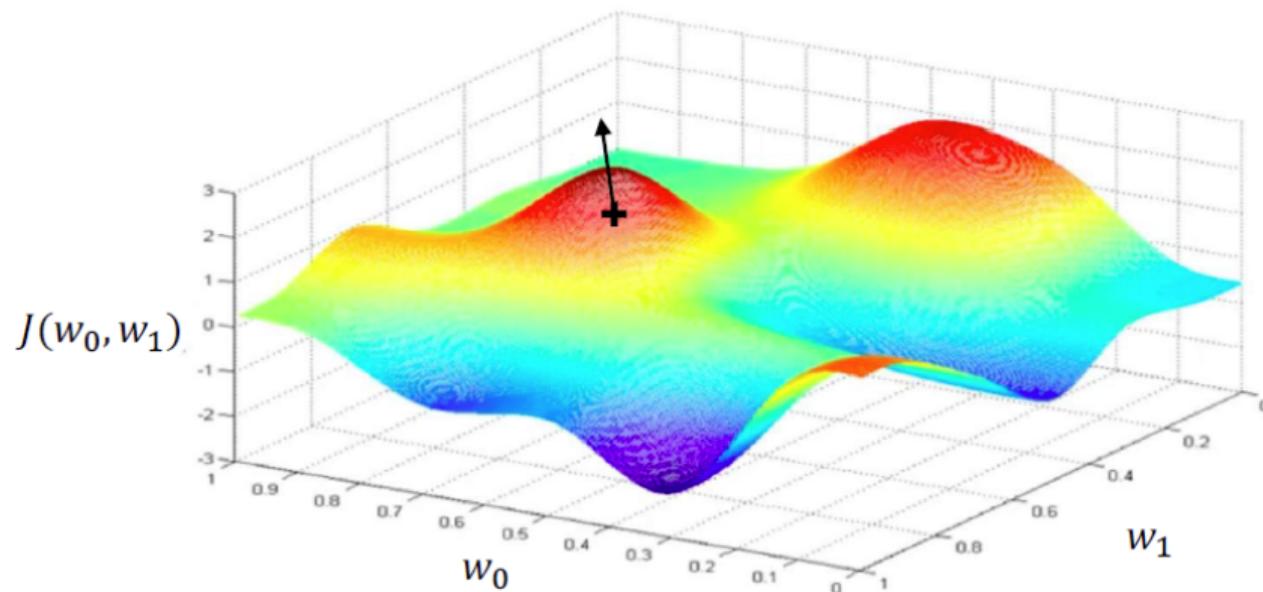
Optimización de la perdida

Escoge al azar una inicial (w_0, w_1)



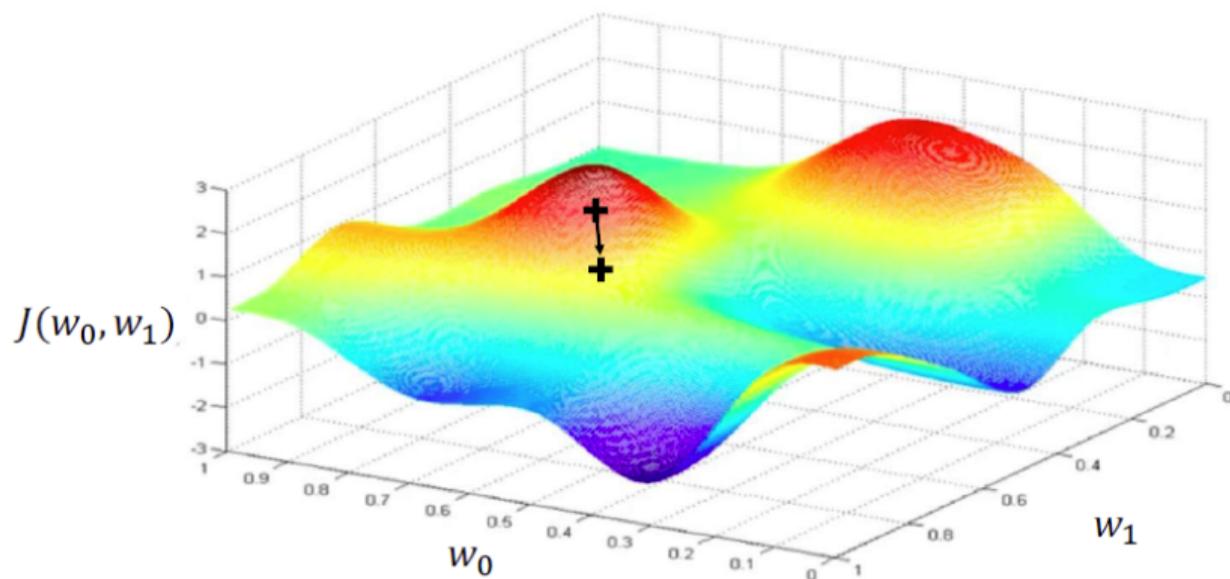
Optimización de la perdida

Calculo del gradiente, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$



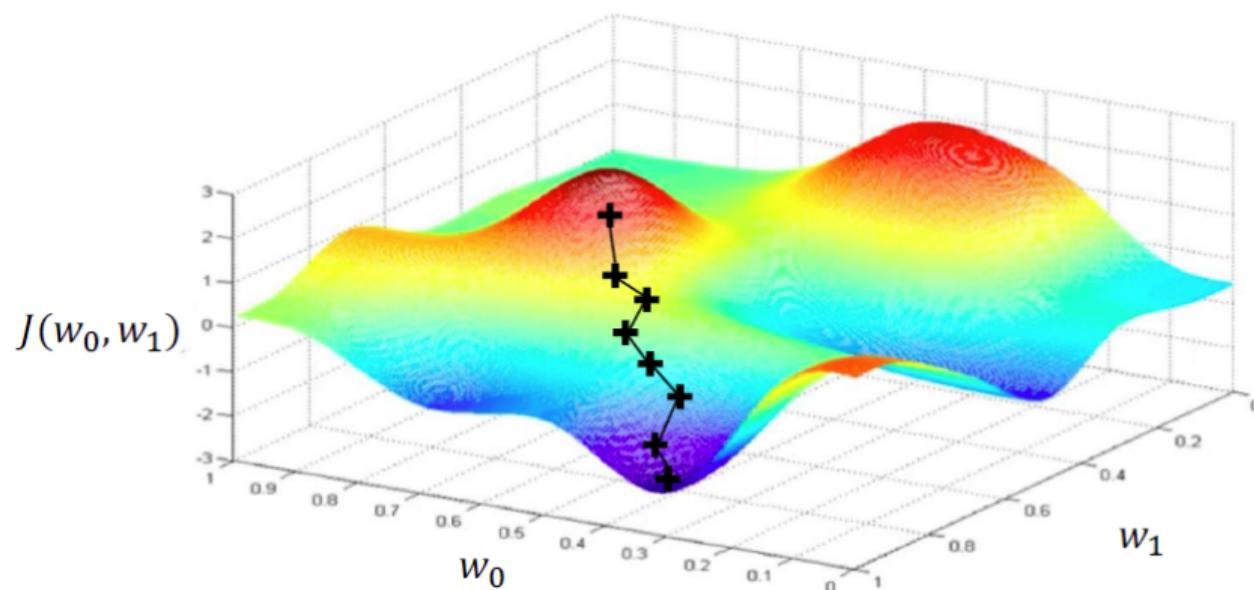
Optimización de la perdida

De un pequeño paso en dirección opuesta al gradiente



Gradiente descendente

Repita hasta la convergencia



Gradiente descendente

Algoritmo:

- Iniciar los pesos al azar $\sim \mathcal{N}(0, \sigma^2)$

```
 weights = tf.random_normal(shape, stddev=sigma)
```

- Bucle hasta la convergencia:

- Calcular el gradiente $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$

```
 grads = tf.gradients(y=loss, xs=weights)
```

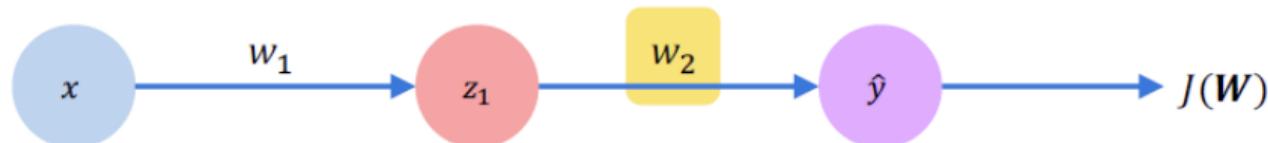
- Actualizar los pesos $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$

```
 weights_new = weights.assign(weights - lr * grads)
```

- Devuelve los pesos

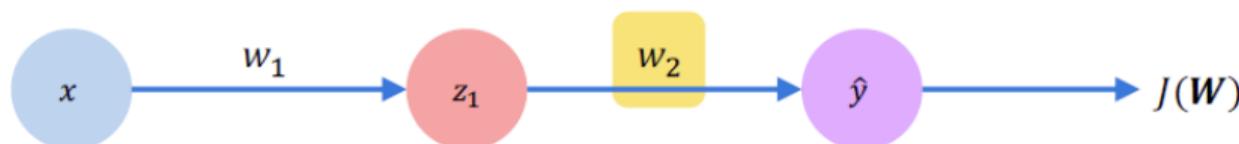
Calculo del gradiente: Propagación hacia atrás

Backpropagation



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

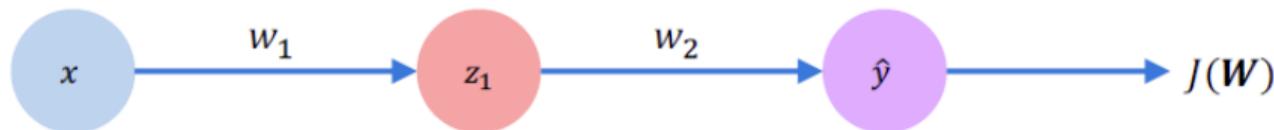
Calculo del gradiente: Propagación hacia atrás



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

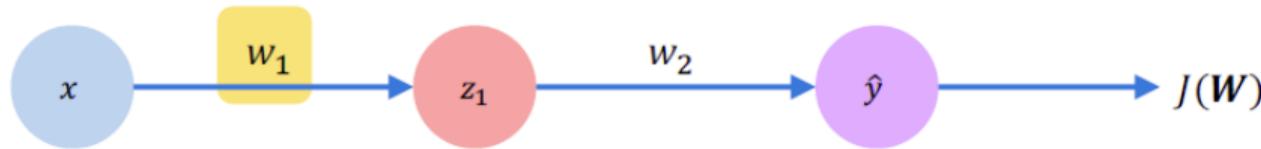
Aplicamos la regla de la cadena

Calculo del gradiente: Propagación hacia atrás



$$\frac{\partial J(\mathbf{W})}{\partial w_2} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

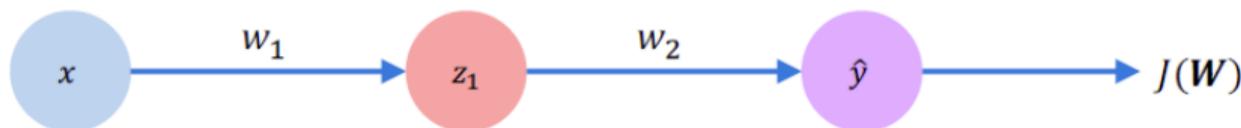
Calculo del gradiente: Propagación hacia atrás



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

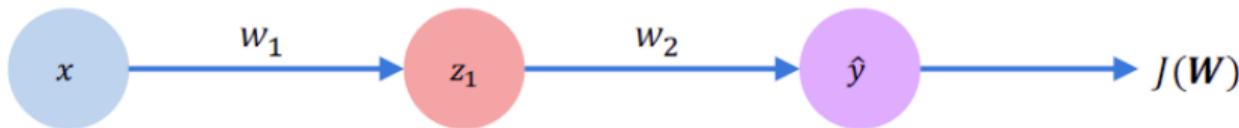
↑ — ↑
 Aplicar regla de la cadena Aplicar regla de la cadena

Calculo del gradiente: Propagación hacia atrás



$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Calculo del gradiente: Propagación hacia atrás

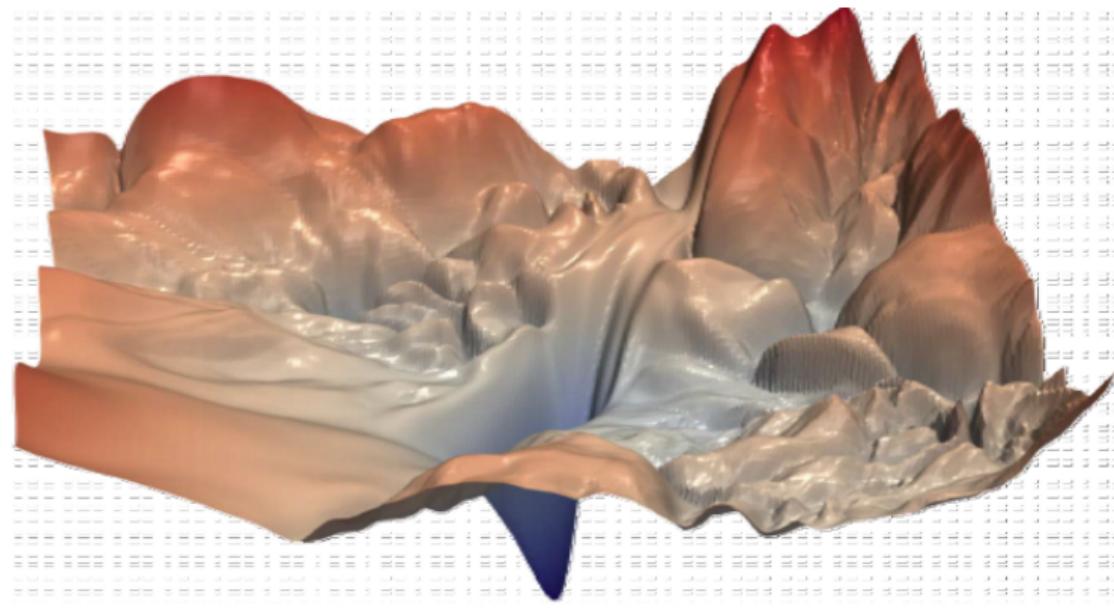


$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

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

Redes neuronales en la práctica: Optimización

El entrenamiento de redes neuronales es complejo



"Visualizando el paisaje de pérdida de las redes neuronales". Diciembre de 2017.

Las funciones de pérdida pueden ser difíciles de optimizar

Recuerde: Optimización a través del descenso del gradiente

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

Las funciones de pérdida pueden ser difíciles de optimizar

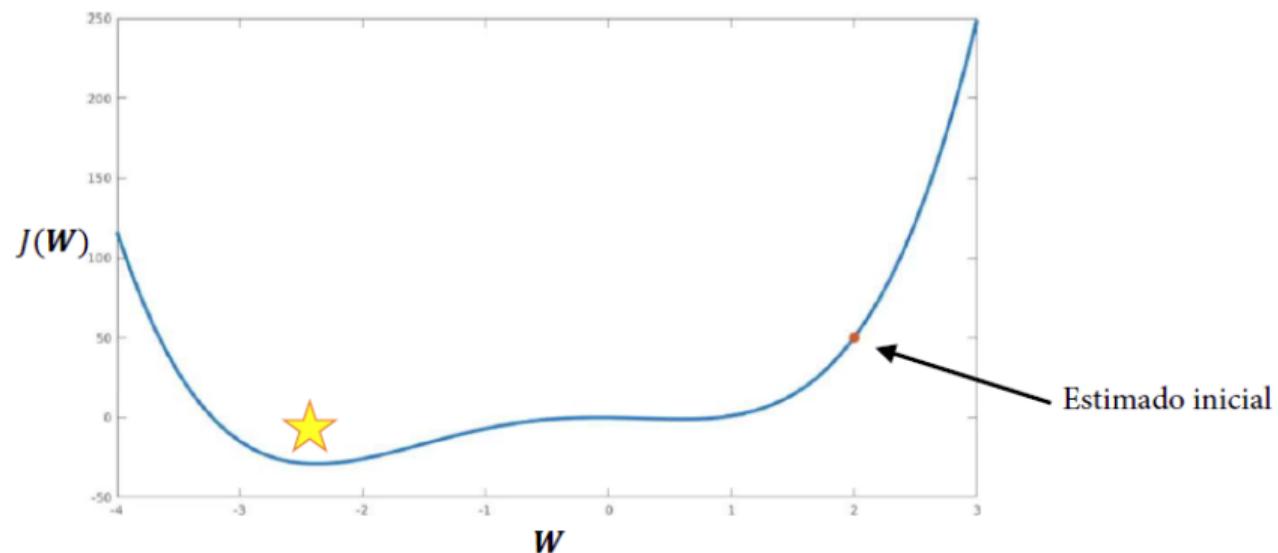
Recuerde: Optimización a través del descenso del gradiente

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

¿Cómo podemos establecer la tasa de aprendizaje? (*learning rate*)

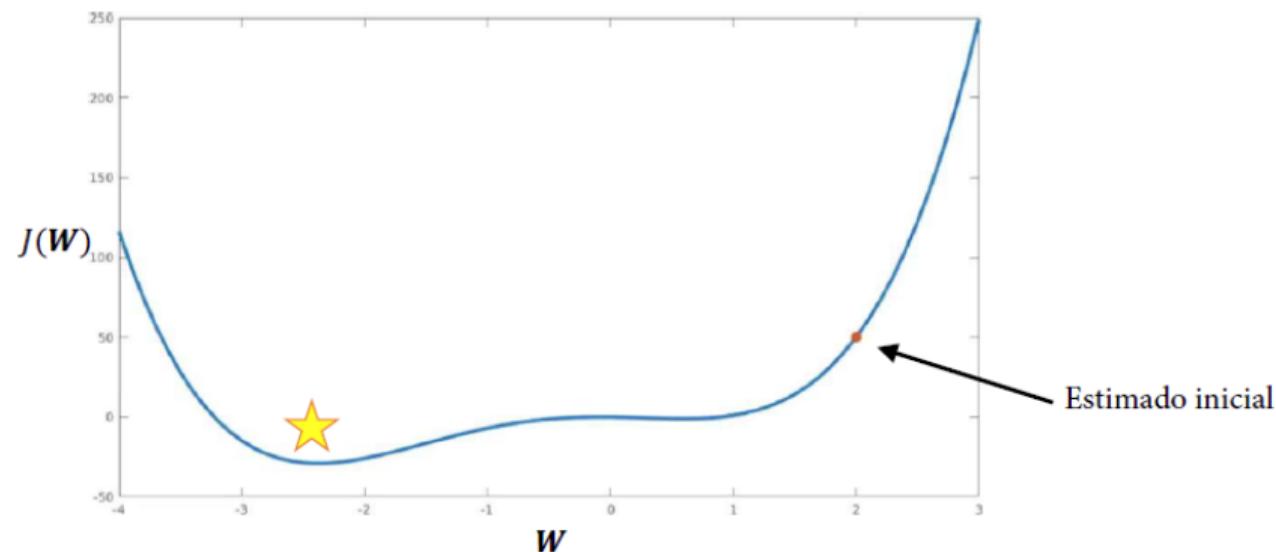
Ajuste de la tasa de aprendizaje

Recuerde: Una pequeña tasa de aprendizaje converge lentamente y se atasca en falsos mínimos locales



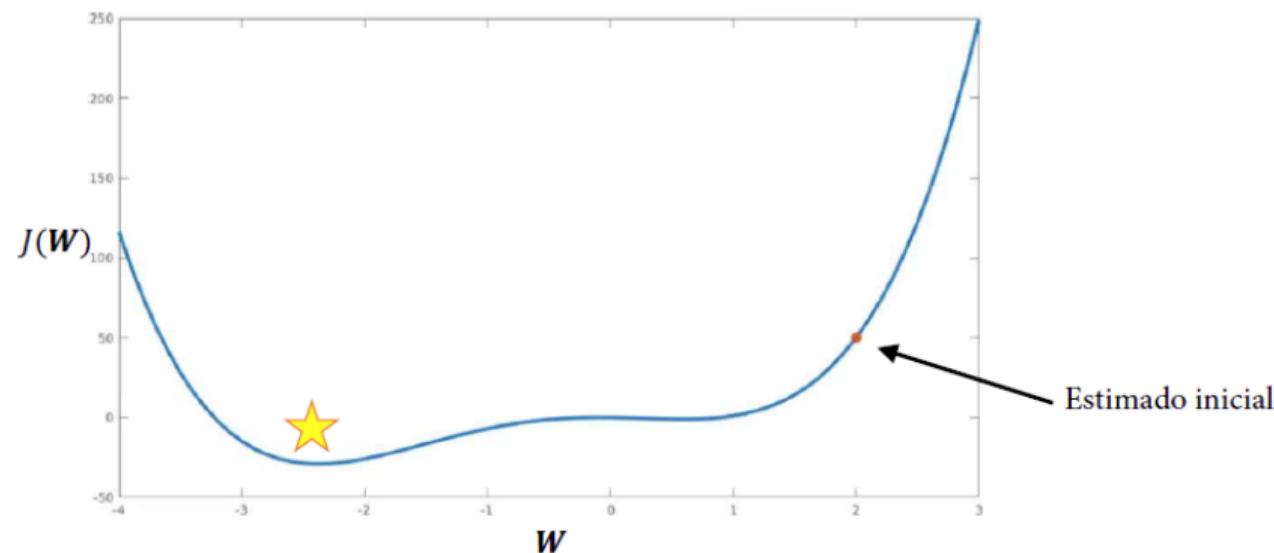
Ajuste de la tasa de aprendizaje

Recuerde: Las grandes tasas de aprendizaje se sobreponen, se vuelven inestables y divergen



Ajuste de la tasa de aprendizaje

Recuerde: Las tasas de aprendizaje estables convergen sin problemas y evitan los mínimos locales



¿Cómo se puede hacer frente a esto?

- **Idea 1:** Intentar muchas tasas de aprendizaje diferentes y ver cuál funciona "bien".
- **Idea 2:** ¡Haz algo más inteligente! Diseñar una tasa de aprendizaje adaptativo 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:
 - de cuán grande sea el gradiente
 - lo rápido que se está aprendiendo
 - tamaño de pesos particulares
 - etc...

Algoritmos de tasas de aprendizaje adaptativas

- Momentum
 - Adagrad
 - Adadelta
 - RMSProp



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

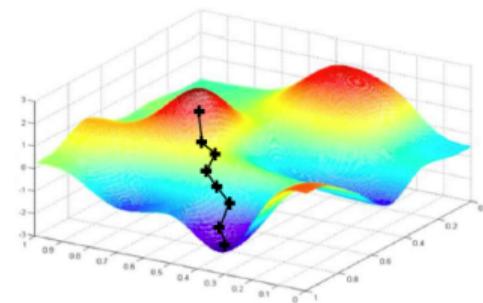
Redes neuronales en la práctica: Mini-lotes

Gradiente descendente

Algoritmo:

- Iniciar los pesos al azar $\sim \mathcal{N}(0, \sigma^2)$
- Bucle hasta la convergencia:
 - Calcular el gradiente $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
 - Actualizar los pesos $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
- Devuelve los pesos

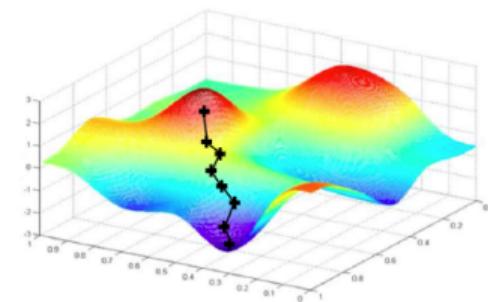
Difícil de calcular $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$



Gradiente descendente estocástico

Algoritmo:

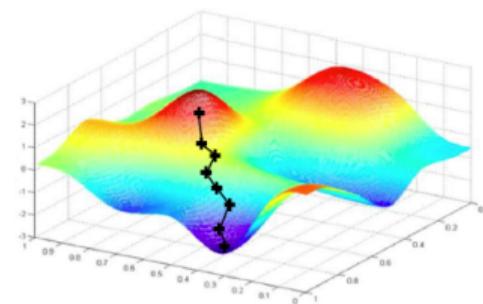
- Iniciar los pesos al azar $\sim \mathcal{N}(0, \sigma^2)$
 - Bucle hasta la convergencia:
 - Tomar un solo punto i
 - Calcular el gradiente $\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}}$
 - Actualizar los pesos $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
 - Devuelve los pesos
- Fácil de calcular $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$ pero muy ruidoso (estocástico)



Gradiente descendente estocástico

Algoritmo:

- Iniciar los pesos al azar $\sim \mathcal{N}(0, \sigma^2)$
 - Bucle hasta la convergencia:
 - Tomar un lote de puntos B
 - Calcular el gradiente $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
 - Actualizar los pesos $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
 - Devuelve los pesos
- Rápido de calcular y una estimación mucho mejor del verdadero gradiente

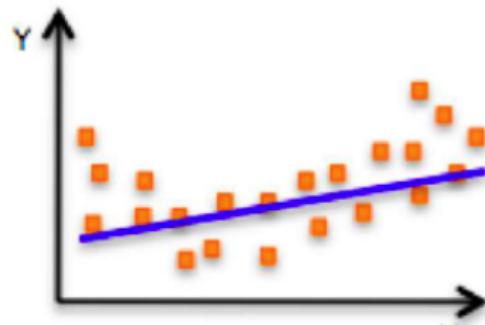


Mini-batches durante el entrenamiento

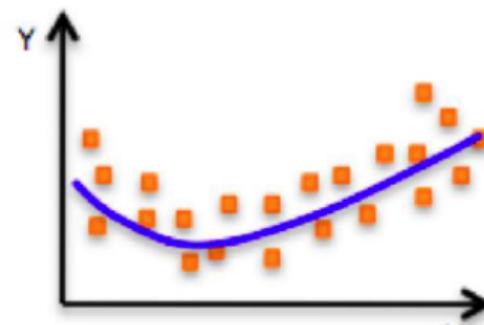
- Estimación más precisa del gradiente:
 - Una convergencia más suave
 - Permite mayores tasas de aprendizaje
- Los mini lotes conducen a un rápido entrenamiento
 - Puede paralelizar la computación + lograr aumentos significativos de velocidad en las GPU

Redes neuronales en la práctica: Sobreajuste (overfitting)

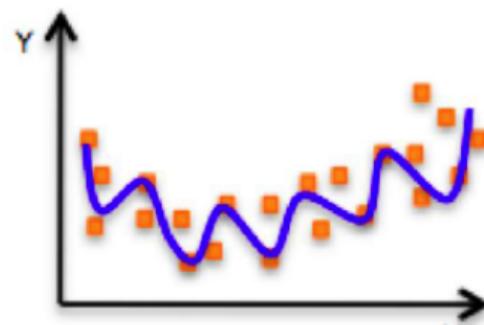
El problema del sobreajuste



Underfitting
El modelo no tiene capacidad para aprender completamente los datos



← → **Ajuste ideal**



Overfitting
Demasiado complejo, parámetros adicionales, no se generaliza bien

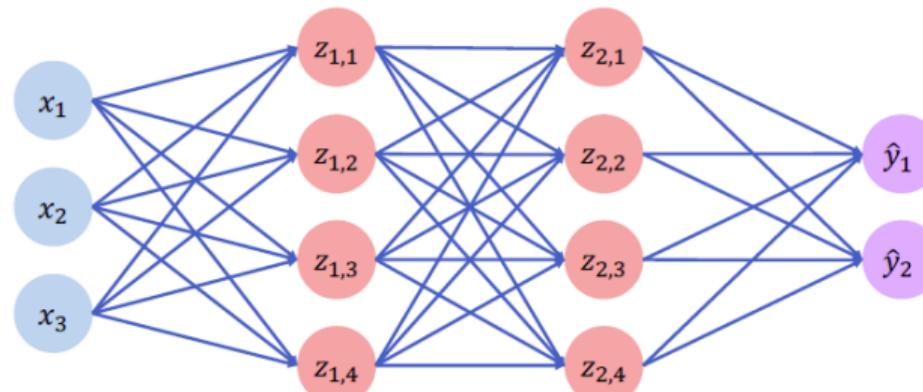
Regularización

- **¿Qué es?** Técnica que limita nuestro problema de optimización para no incentivar la generación de modelos complejos
- **¿Por qué lo necesitamos?** Mejorar la generalización de nuestro modelo sobre datos no vistos

Regularización I: Dropout

Abandono

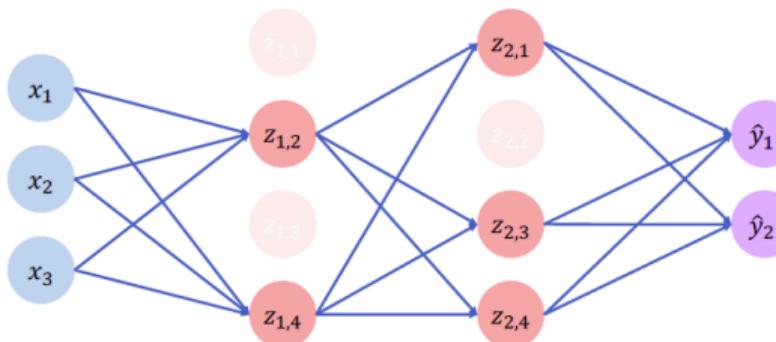
- Durante el entrenamiento, se fijan aleatoriamente algunas activaciones de neuronas en 0



Regularización I: Dropout

Abandono

- Durante el entrenamiento, se fijan aleatoriamente algunas activaciones de neuronas en 0
 - Generalmente se "da de baja" al 50% de las neuronas en una capa
 - Obliga a la red a no depender de ningún nodo

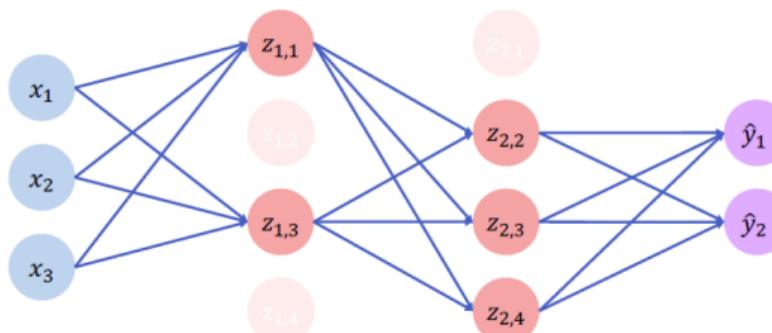


`tf.keras.layers.Dropout (p=0.5)`

Regularización I: Dropout

Abandono

- Durante el entrenamiento, se fijan aleatoriamente algunas activaciones de neuronas en 0
 - Generalmente se "da de baja" al 50% de las neuronas en una capa
 - Obliga a la red a no depender de ningún nodo



tf.keras.layers.Dropout(p=0.5)

Regularización II: Parada temprana

Early Stopping

- Detener el entrenamiento antes de empezar a sobreajustar...



Regularización II: Parada temprana

- Detener el entrenamiento antes de empezar a sobreajustar...



Regularización II: Parada temprana

- Detener el entrenamiento antes de empezar a sobreajustar...



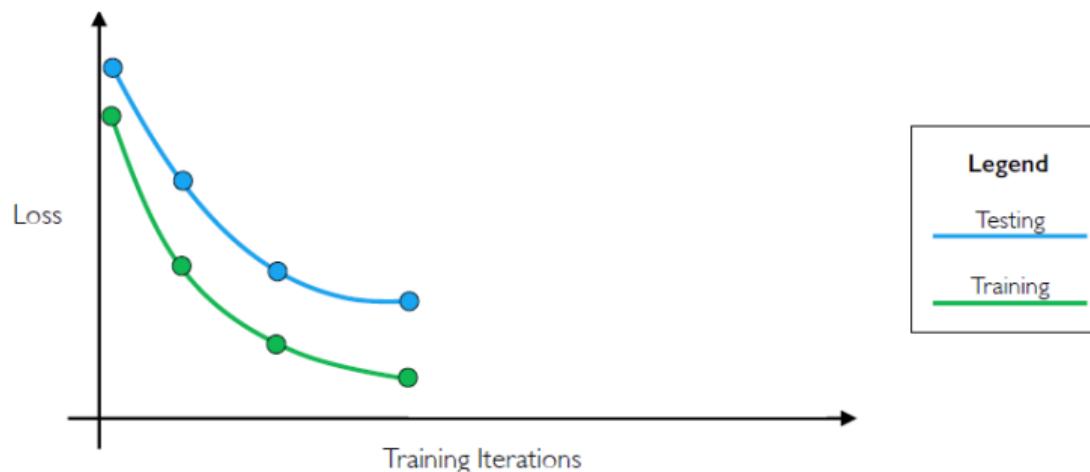
Regularización II: Parada temprana

- Detener el entrenamiento antes de empezar a sobreajustar...



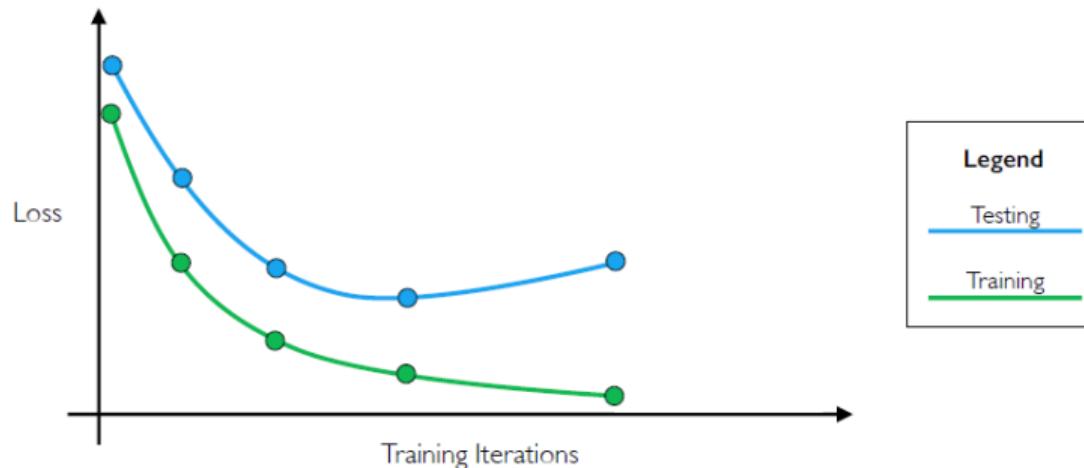
Regularización II: Parada temprana

- Detener el entrenamiento antes de empezar a sobreajustar...



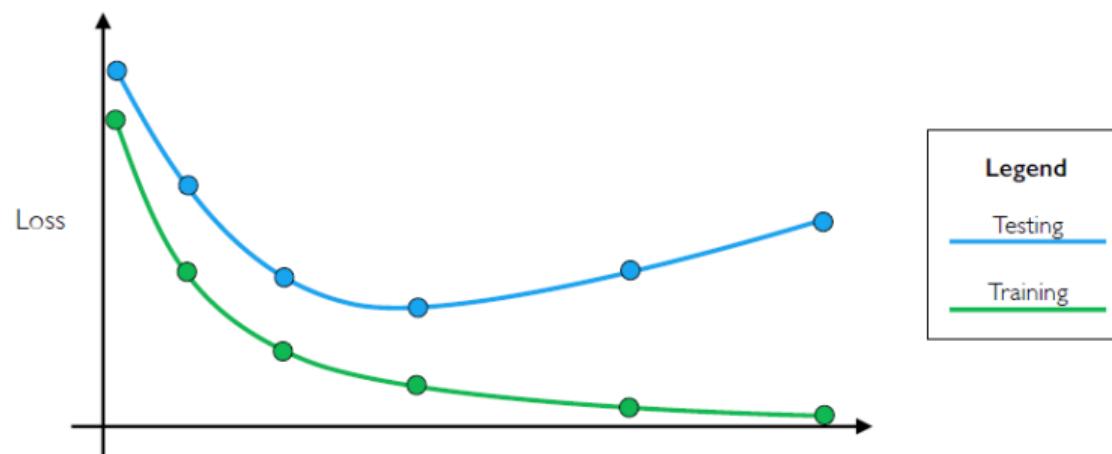
Regularización II: Parada temprana

- Detener el entrenamiento antes de empezar a sobreajustar...



Regularización II: Parada temprana

- Detener el entrenamiento antes de empezar a sobreajustar...



Regularización II: Parada temprana

- Detener el entrenamiento antes de empezar a sobreajustar...



Regularización II: Parada temprana

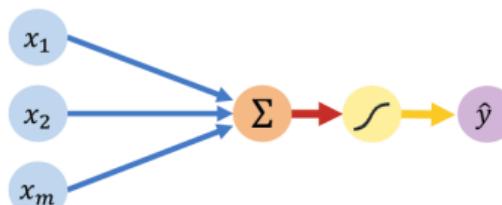
- Detener el entrenamiento antes de empezar a sobreajustar...



Review

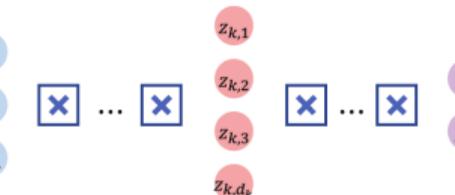
El Perceptrón

- Los bloques de construcción estructurales
- Funciones de activación no lineal



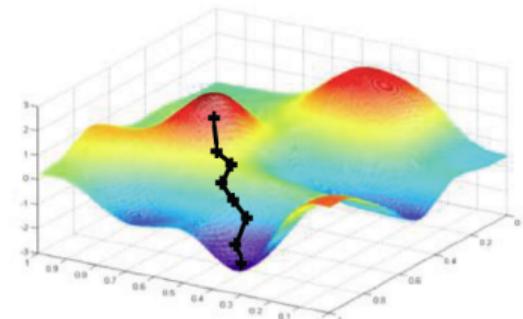
Redes neuronales

- Apilando perceptrones para formar redes neuronales
- Optimización a través de la retropropagación (backpropagation)



Entrenamiento en práctica

- Aprendizaje adaptativo
- *Batching*
- Regularización



Muchas gracias por su atención

¿Preguntas?

Contact: Marco Tulio Teran De La Hoz
e-mail: marco.teran@usa.edu.co

