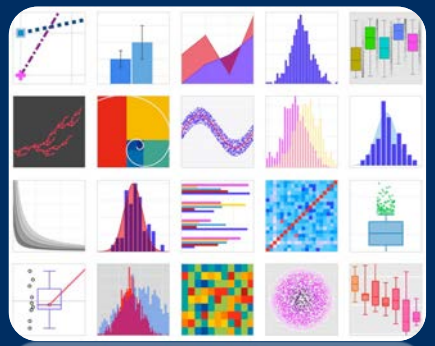
Introducción al aprendizaje de máquinas en Python

Día 5 – Aprendizaje profundo (Deep Learning)



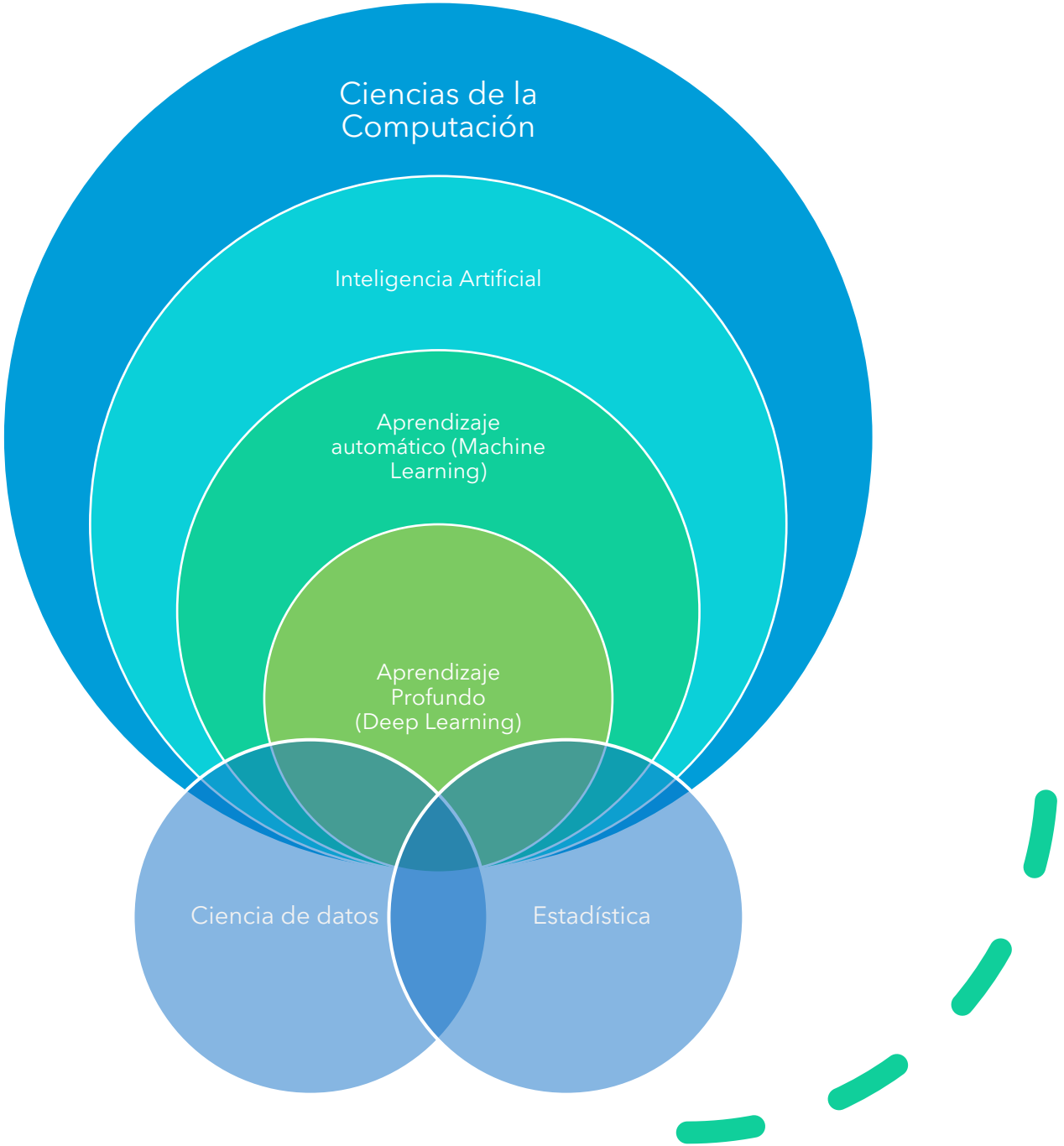
Presentan:
Dr. Ulises Olivares Pinto
Dr. Jesús Emmanuel Solís Pérez
Walter André Rosales Reyes
Escuela Nacional de Estudios Superiores Unidad Juriquilla

Universidad Nacional Autónoma de México

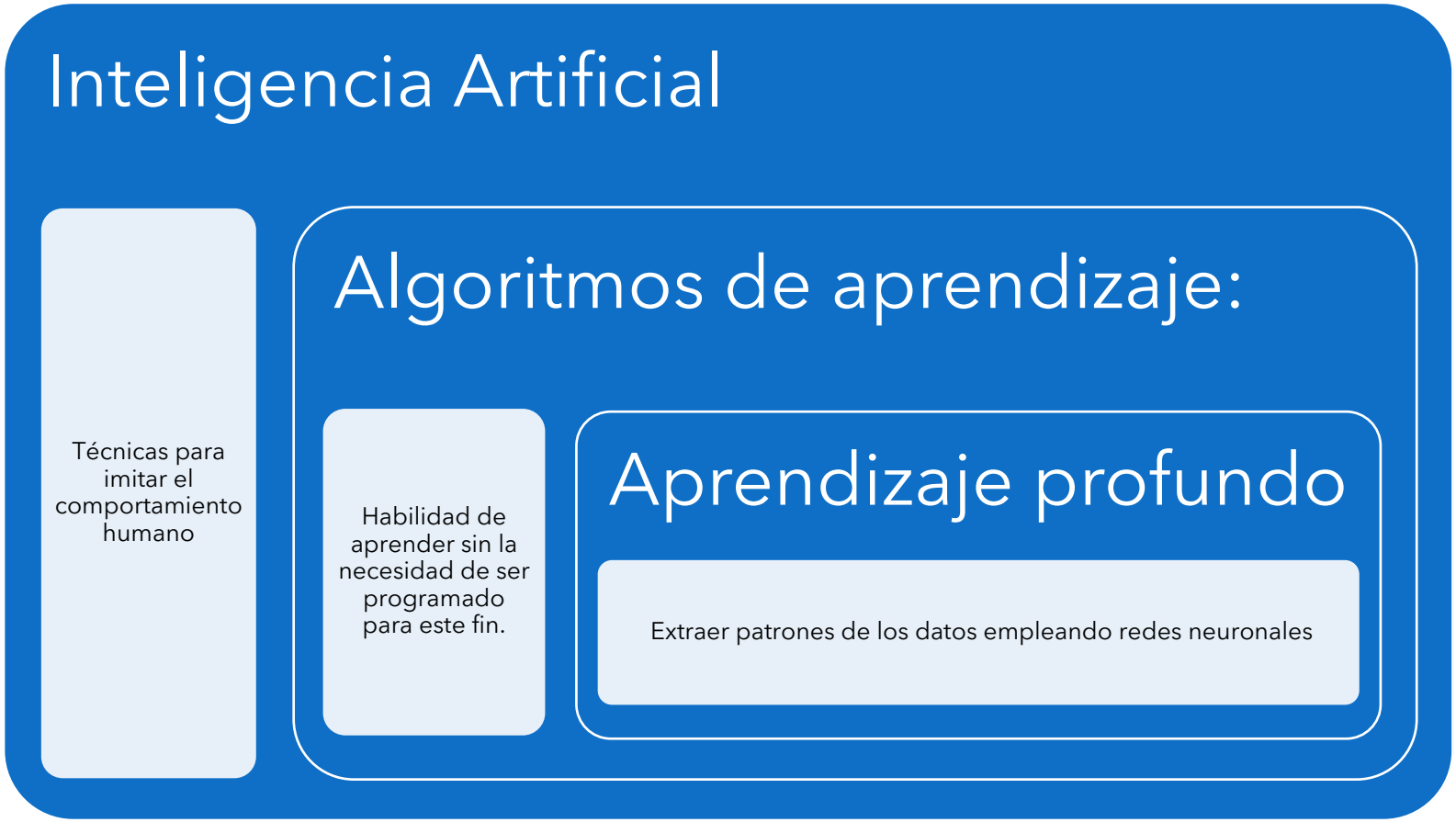
UNAM La Universidad de la Nación

ENES Juriquilla

¿Qué es Deep Learning?



¿Qué es Deep Learning?



- **Algoritmos de aprendizaje:**
  - Definen un conjunto de características a partir de los datos
    - Proceso manual (en muchas ocasiones)
    - Se deben especificar las variables de interés.

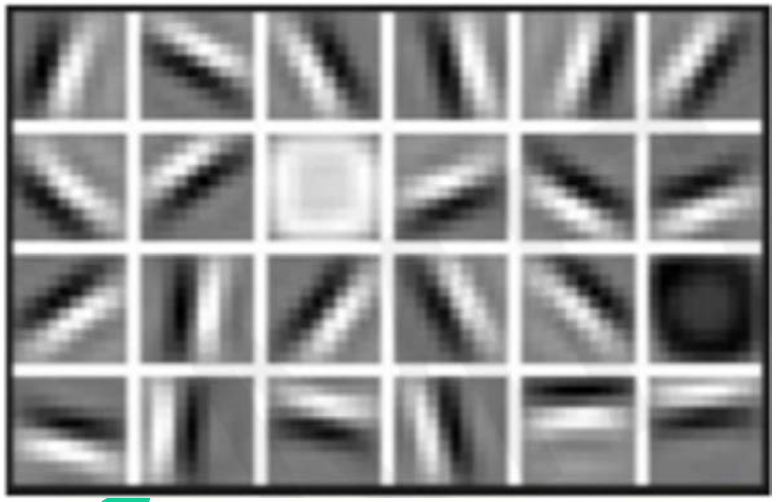
- **Deep Learning:**
  - Emplea técnicas para aprender características directamente de los datos.

¿Por qúe Deep learning?

¿Por qúe Deep learning?

- Extraer características directamente de los datos
  - Caso de estudio: reconocimiento facial

Bajo nivel



Líneas y aristas

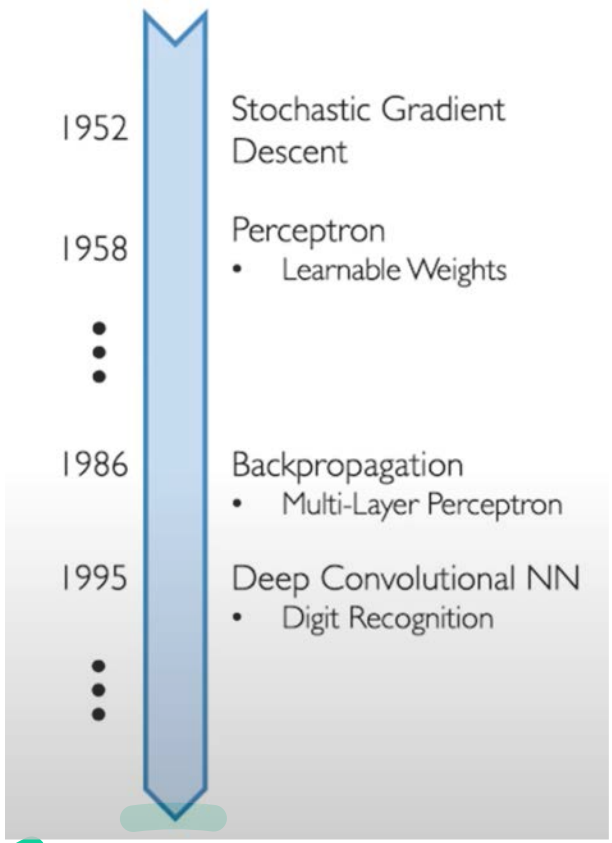
Nivel medio

Características faciales
Nariz, boca, ojos

Alto nivel

Estructura facial

¿Por qué ahora?



1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET

WIKIPEDIA
The Free Encyclopedia

2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software

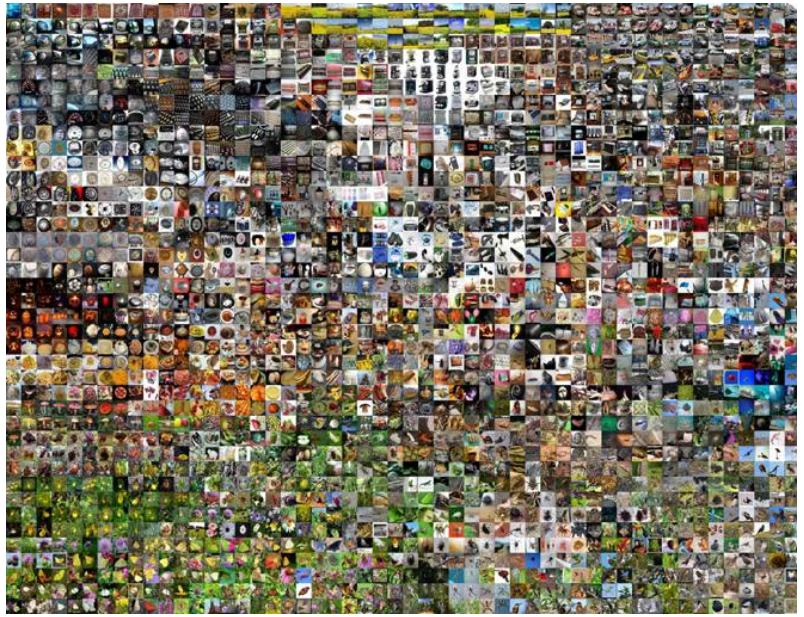
- Improved Techniques
- New Models
- Toolboxes

TensorFlow

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Imagenet

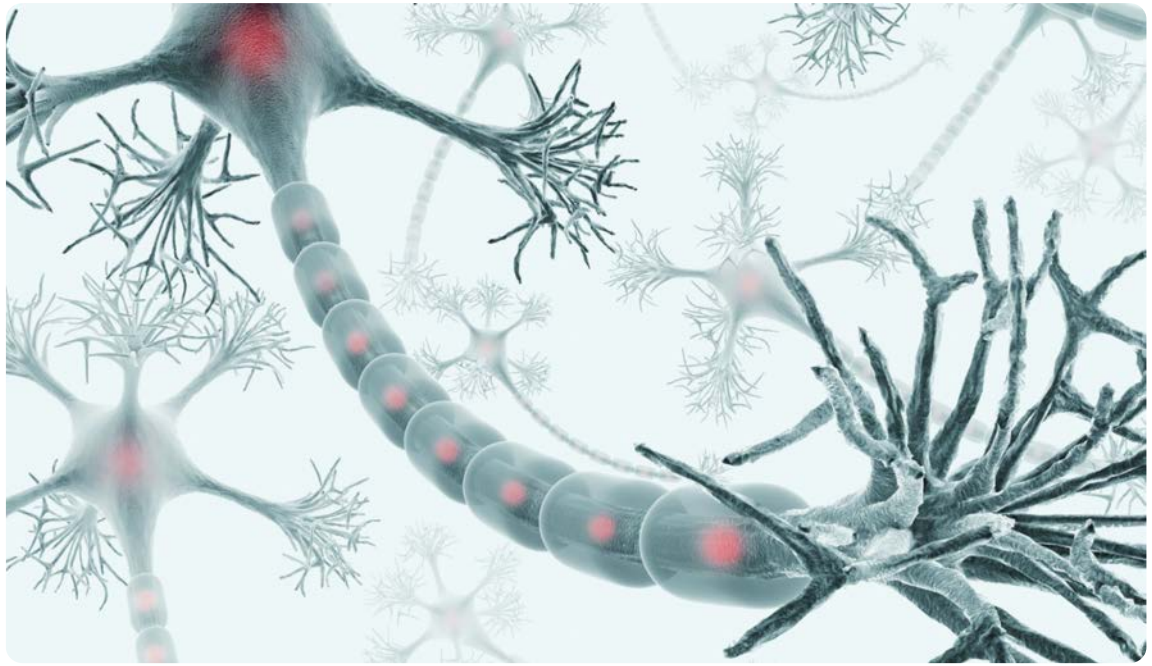
http://www.image-net.org/

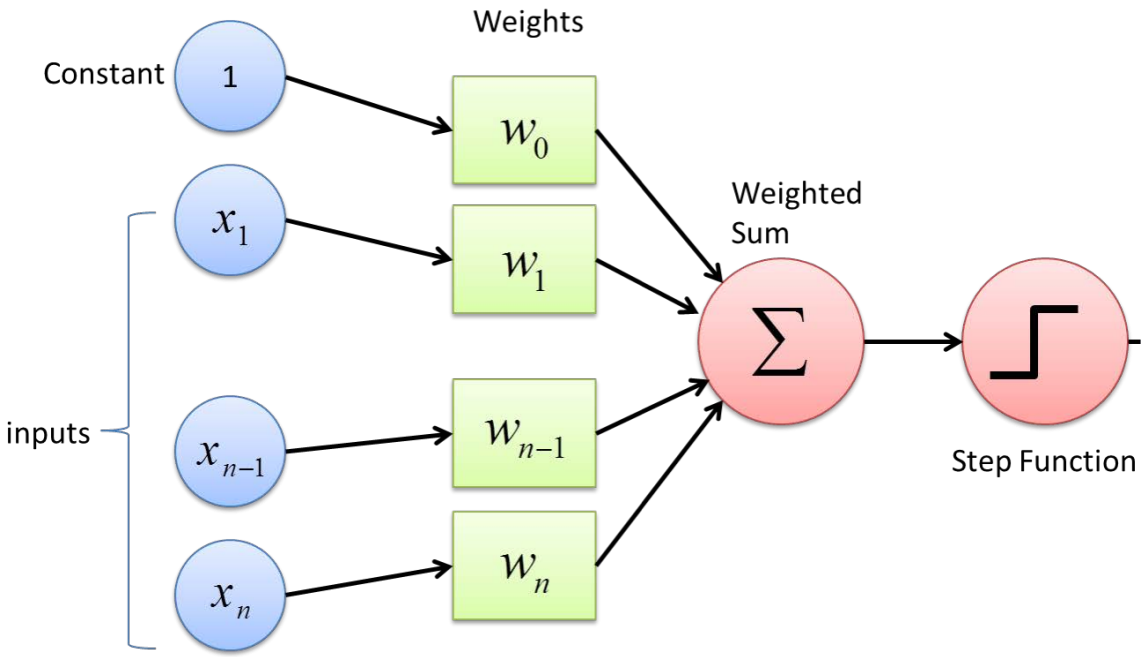


Fuente:Olga Russakovsky*, Jia Deng*, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei. (* = equal contribution) **ImageNet Large Scale Visual Recognition Challenge**. *IJCV,* 2015.

Perceptrón

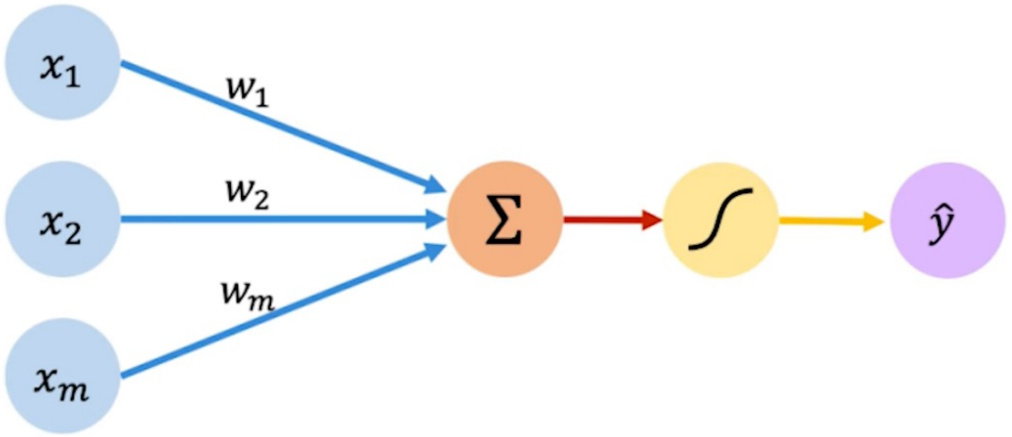
El modelo fundamental (básico) para Deep Learning

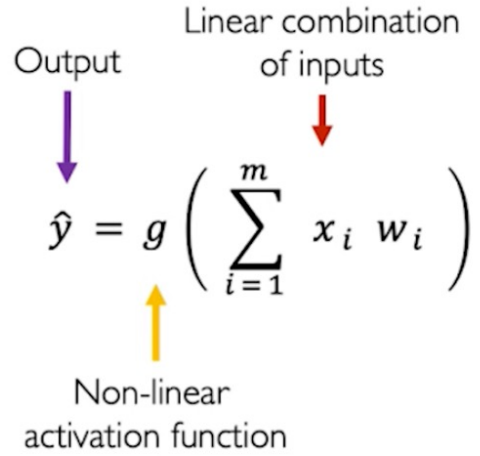




Perceptrón: Propagación hacia delante (forward propagation)

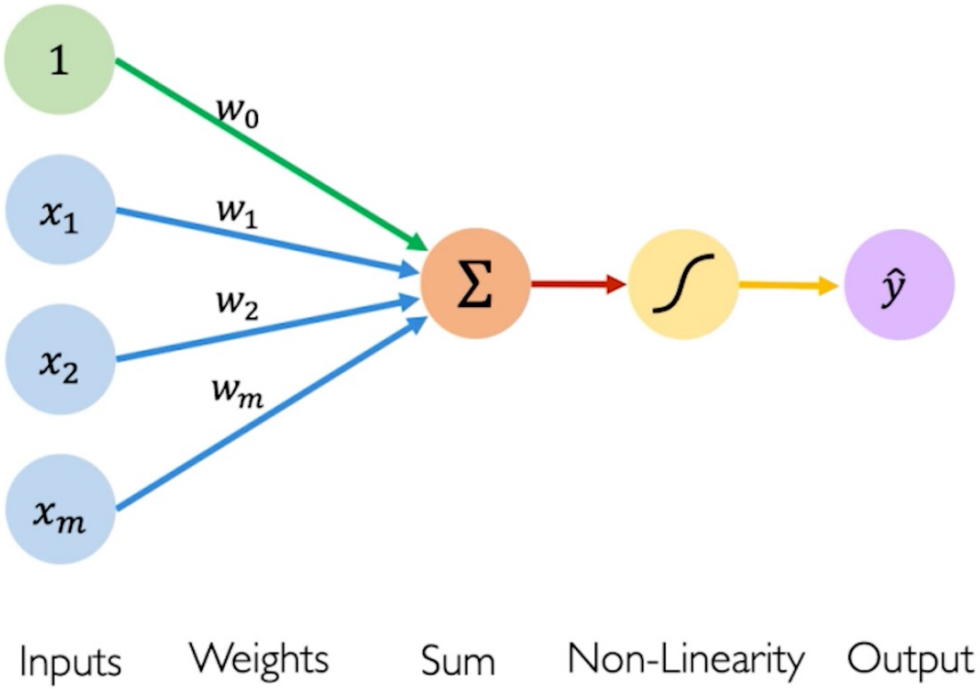
Una neurona simple

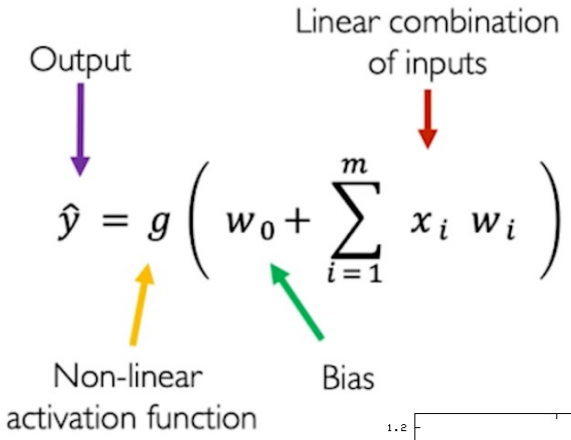


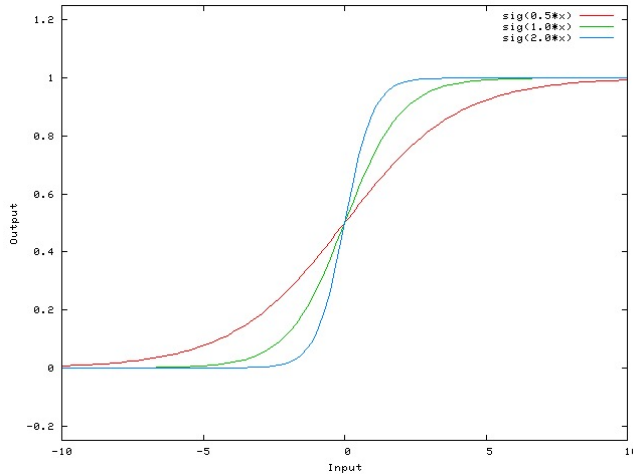


Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Perceptrón: Propagación hacia delante (forward propagation)

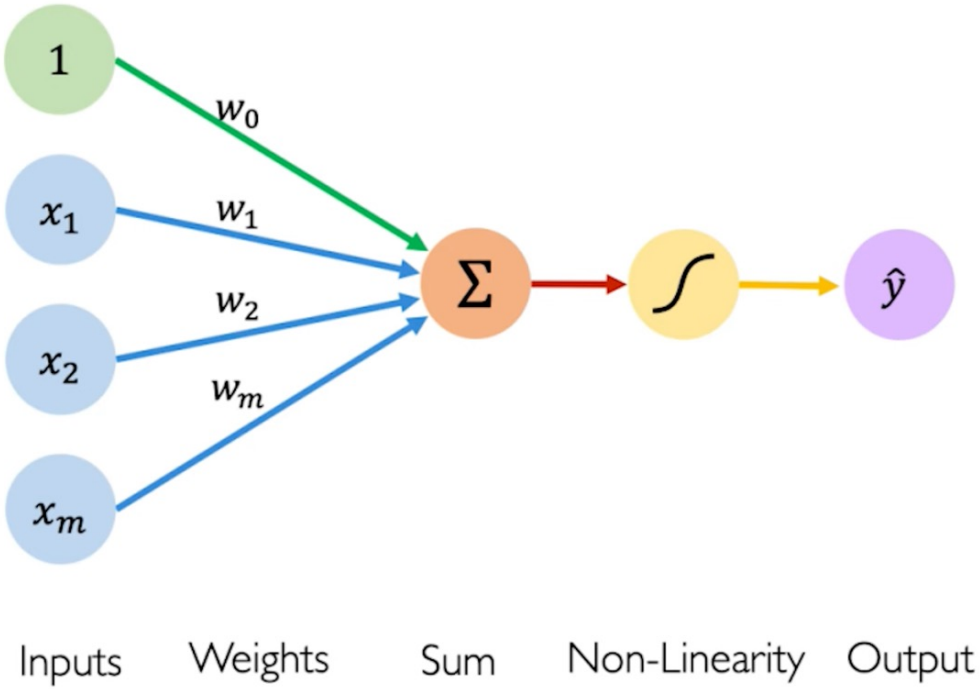


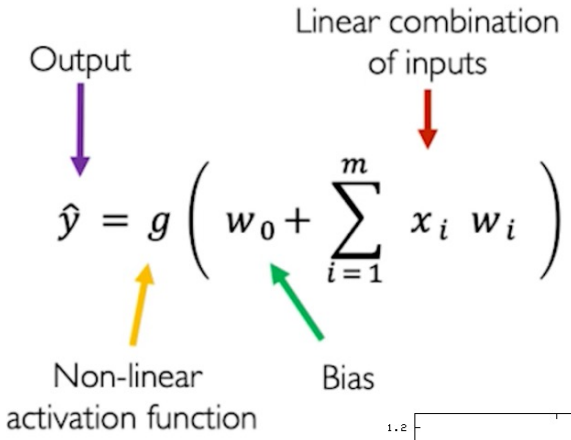


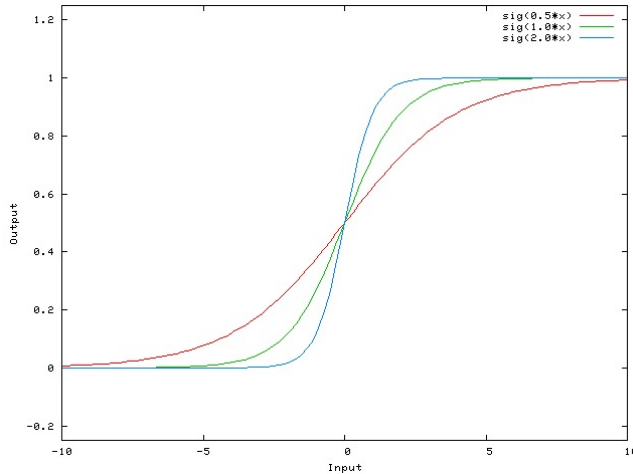


Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Perceptrón: Propagación hacia delante (forward propagation)

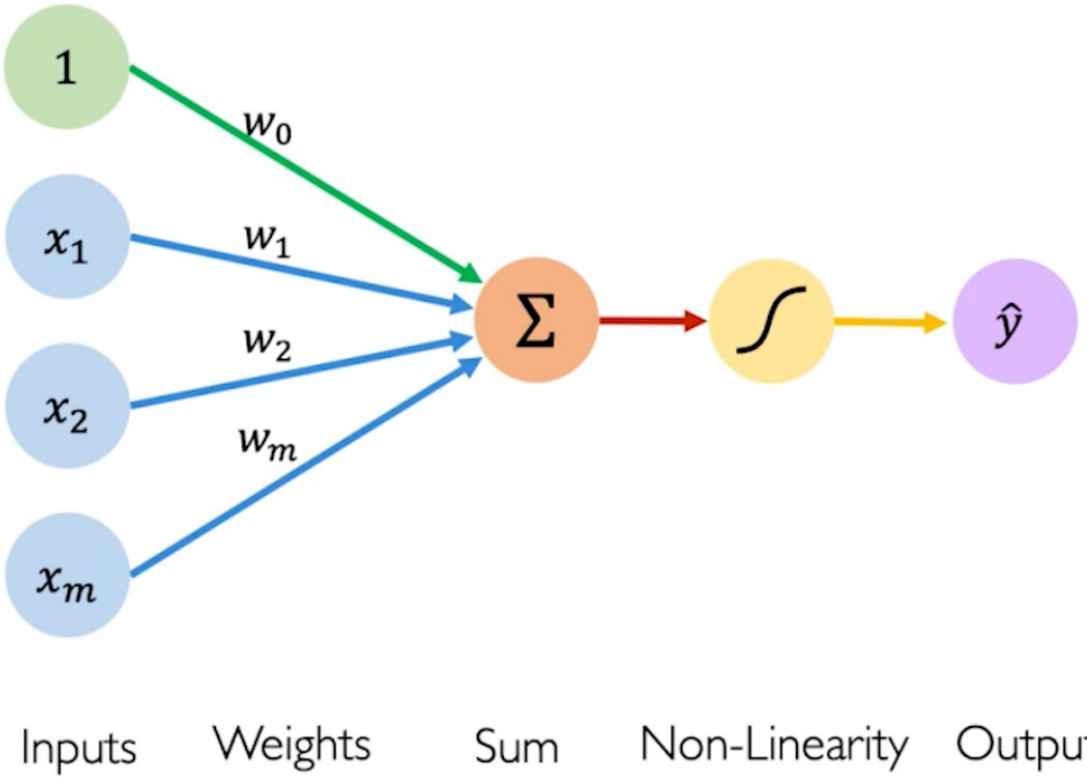






Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Perceptrón: Propagación hacia delante - bias (forward propagation)



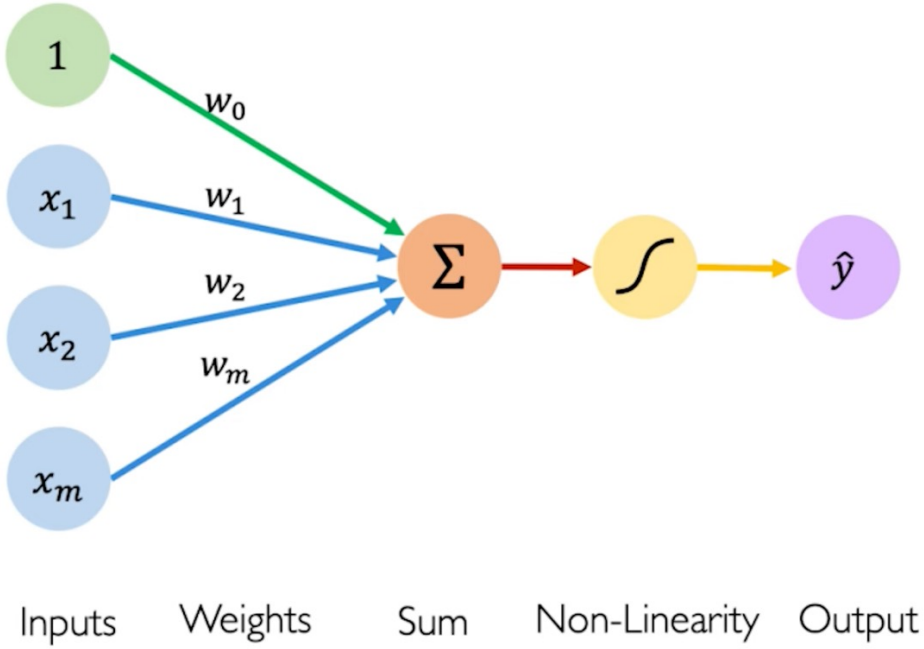
$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

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

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

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Perceptrón: Propagación hacia delante - bias (forward propagation)

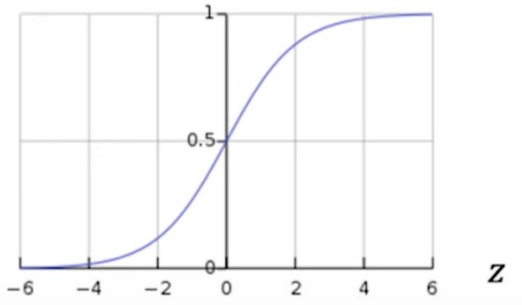


Activation Functions

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

- Example: sigmoid function

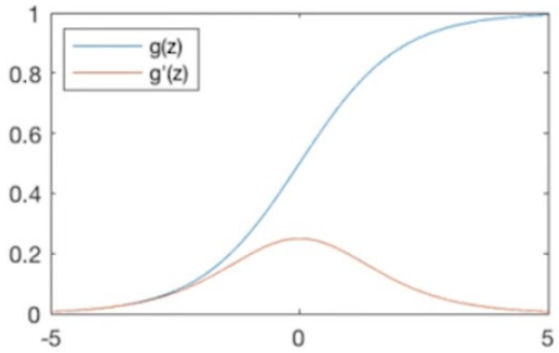
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Funciones más comunes de activación (no lineales)

Sigmoid Function

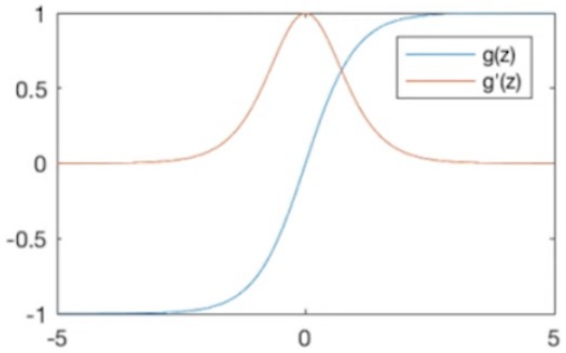


$$g(z) = \frac{1}{1 + e^{-z}}$$

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

`tf.math.sigmoid(z)`

Hyperbolic Tangent

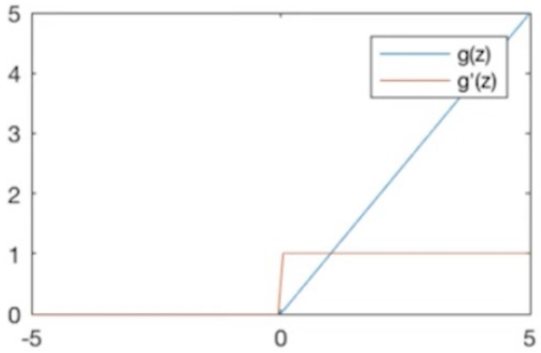


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

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

`tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



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

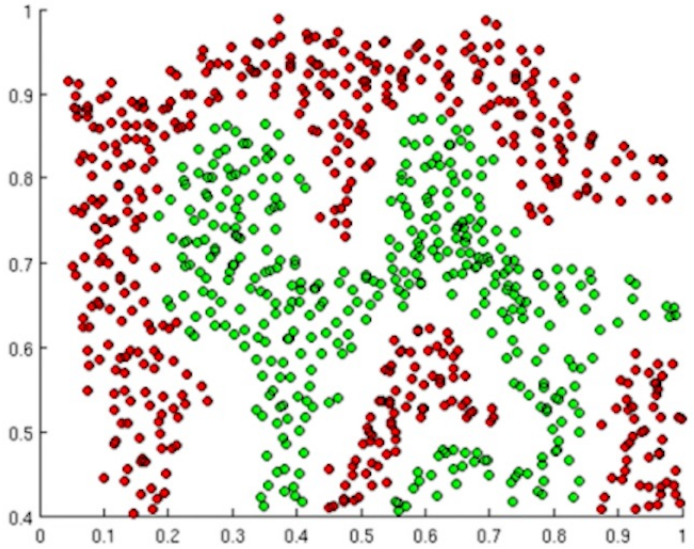
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

`tf.nn.relu(z)`

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

¿Por qué Funciones de activación?

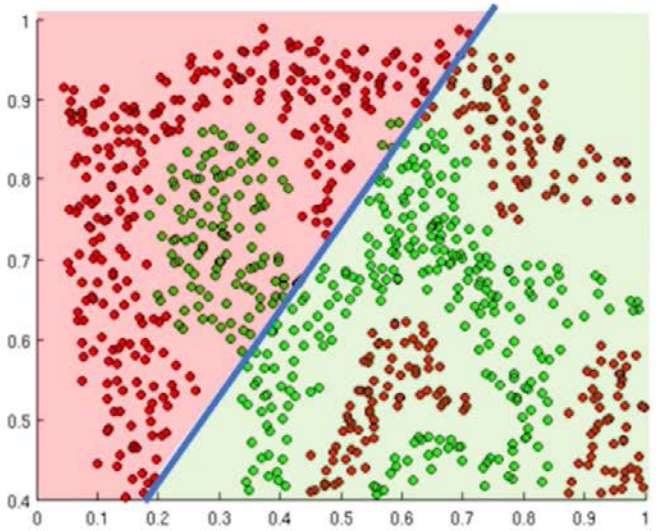
- Introducen no linealidad en la Red Neuronal (RN).
  - Problema: Es necesario diferenciar los puntos rojos de los puntos verdes. Se deberá usar una función lineal.

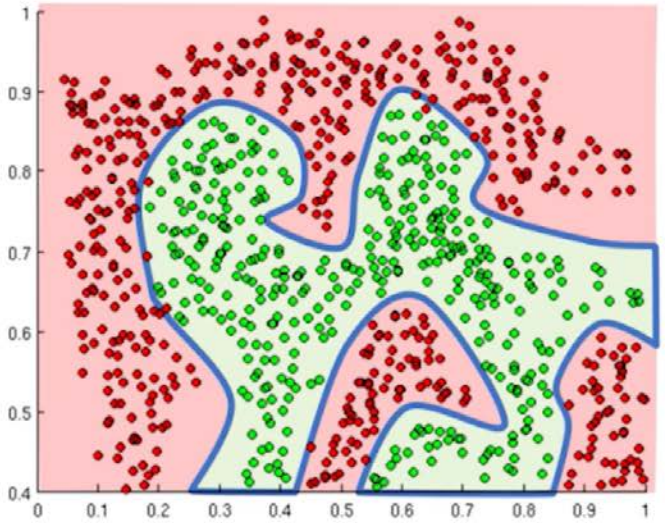


Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

¿Por qué Funciones de activación?

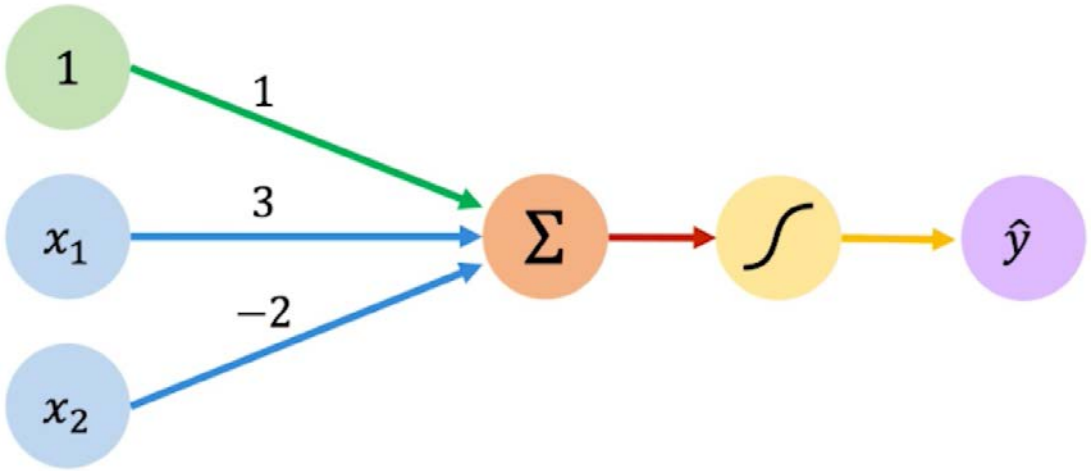
- Introducen no linealidad en la Red Neuronal (RN).
  - Problema: Es necesario diferenciar los puntos rojos de los puntos verdes. Se deberá usar una función lineal.





Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Perceptrón: Ejemplo

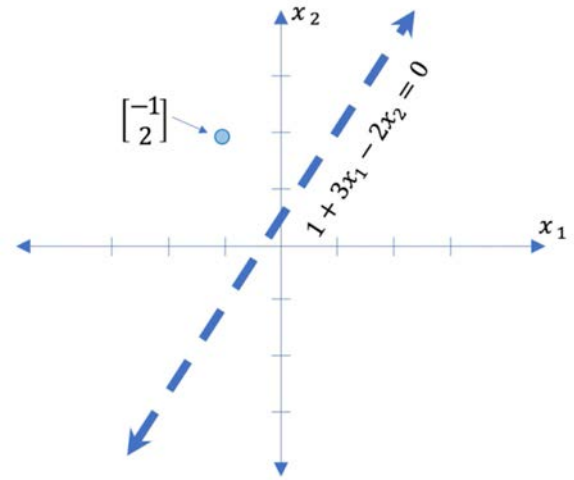


$$w_0 = 1 \text{ and } \boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$$

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

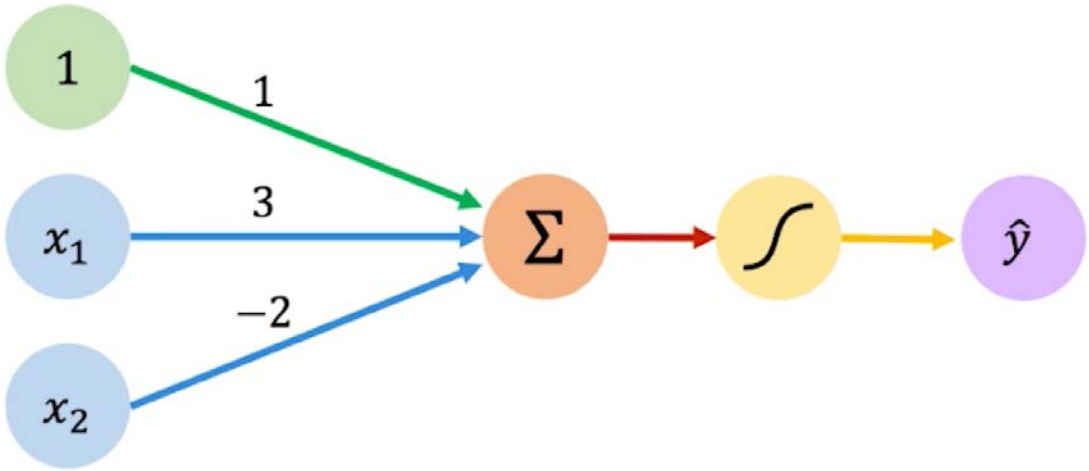
$$\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$

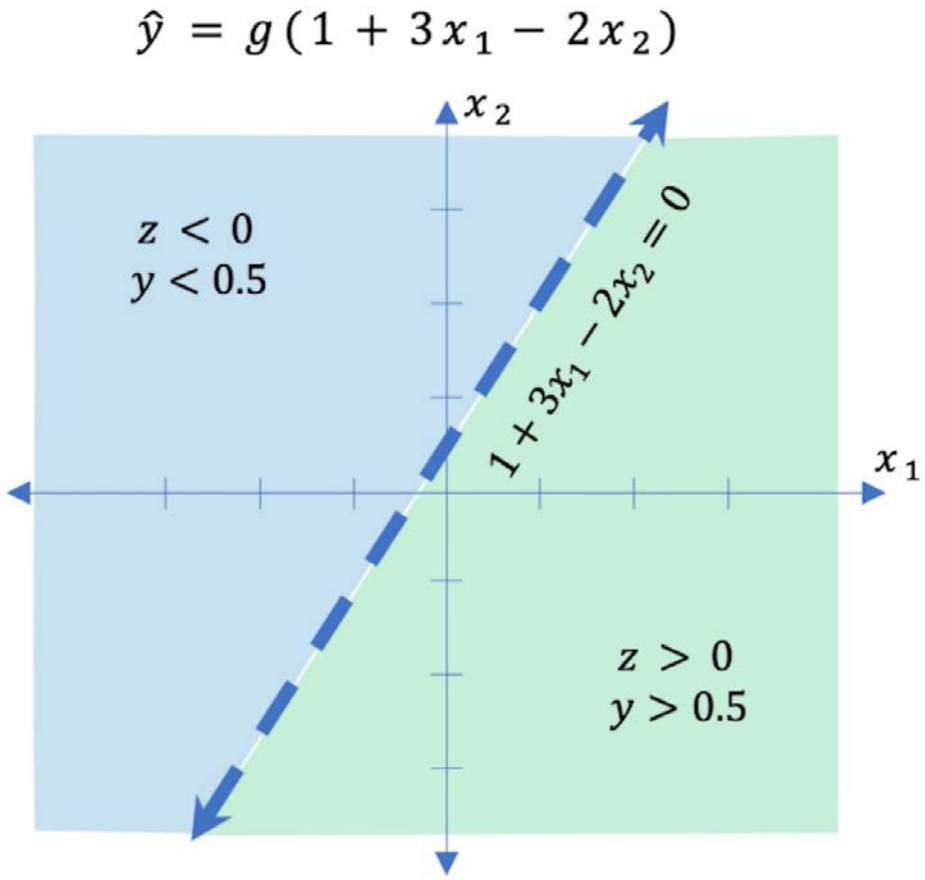
$$\hat{y} = g\left(1 + (3 * -1) - (2 * 2)\right)$$
$$= g(-6) \approx 0.002$$



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Perceptrón: Ejemplo



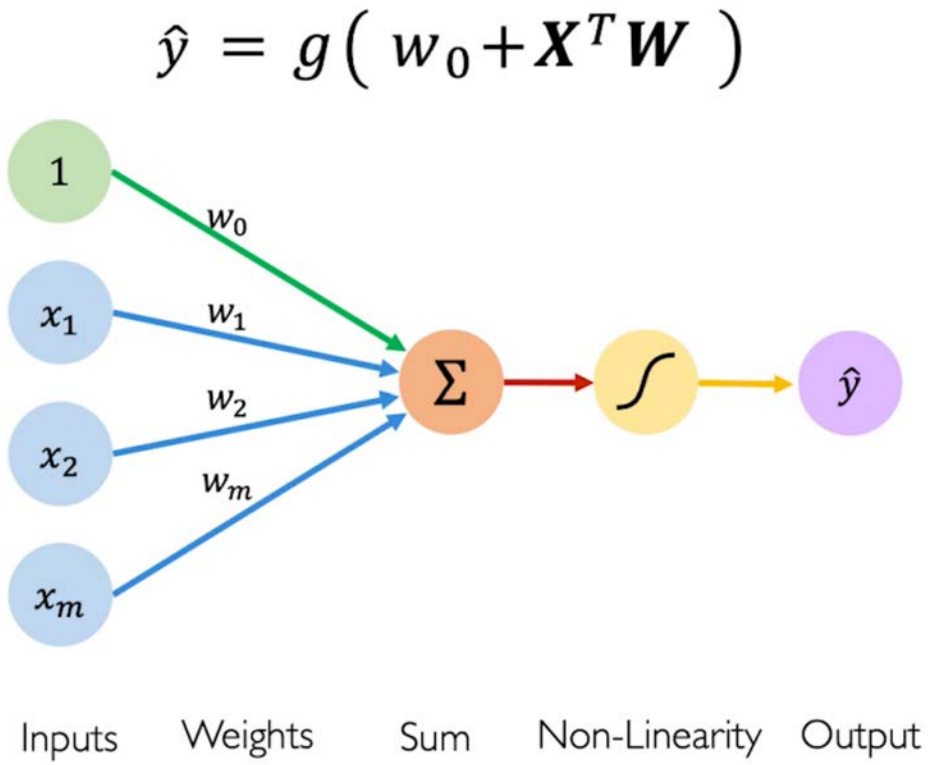


Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Perceptrón Simplificado

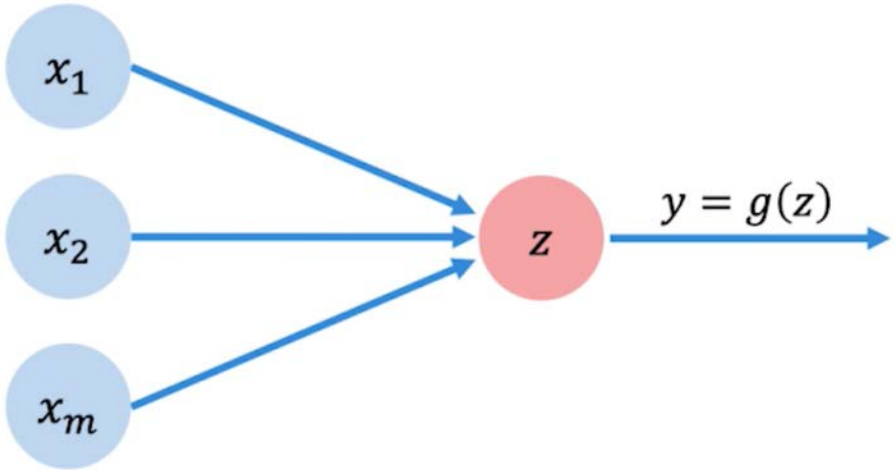
Tres pasos importantes:

1. Producto punto
2. Bias (sesgo)
3. Función no lineal



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

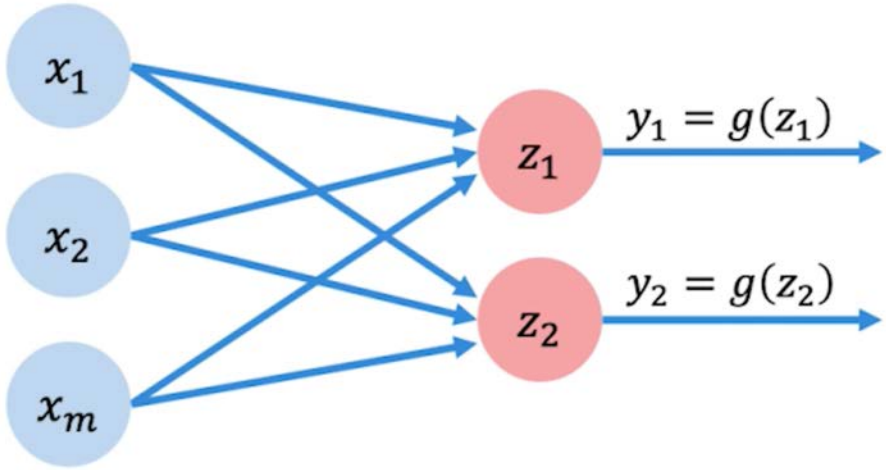
Perceptrón Simplificado



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

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Perceptrón con múltiples salidas



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

Debido a que todas las entradas están densamente conectadas con todas las salidas. A estas capas se les denomina capas densas.

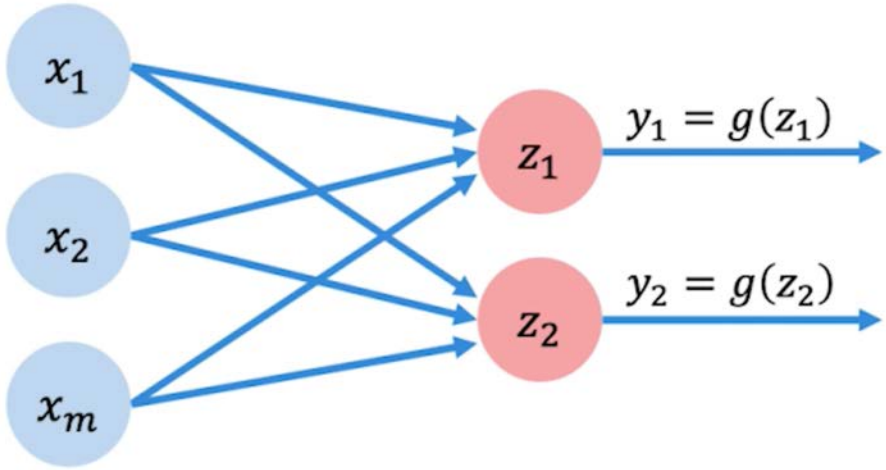
Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Capas densas
Ejemplo de implementación

```python
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
```

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Perceptrón con múltiples salidas

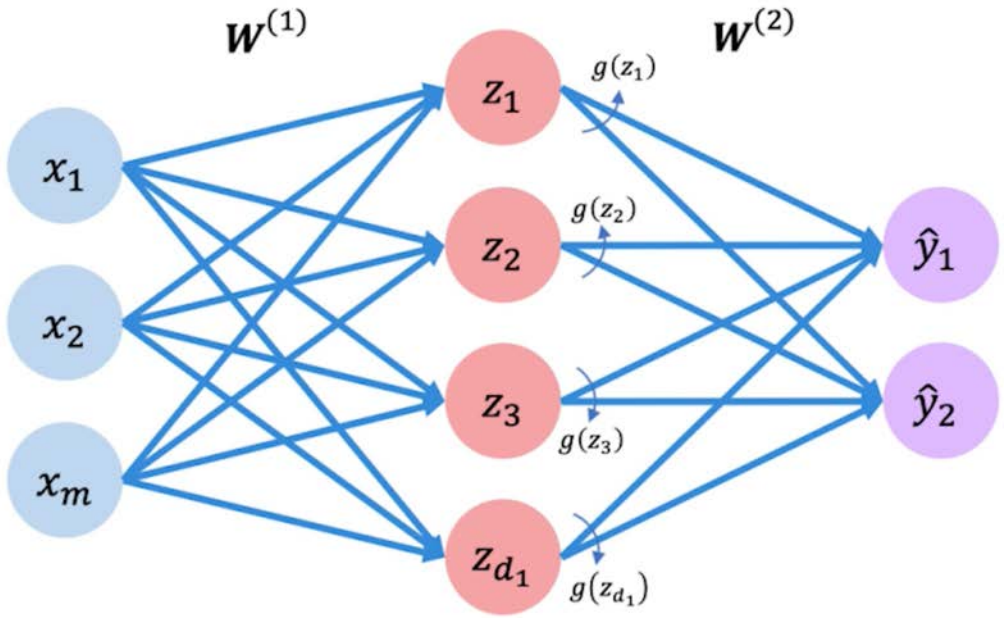



```
import tensorflow as tf

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

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

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Red Neuronal con una sola Capa (oculta)

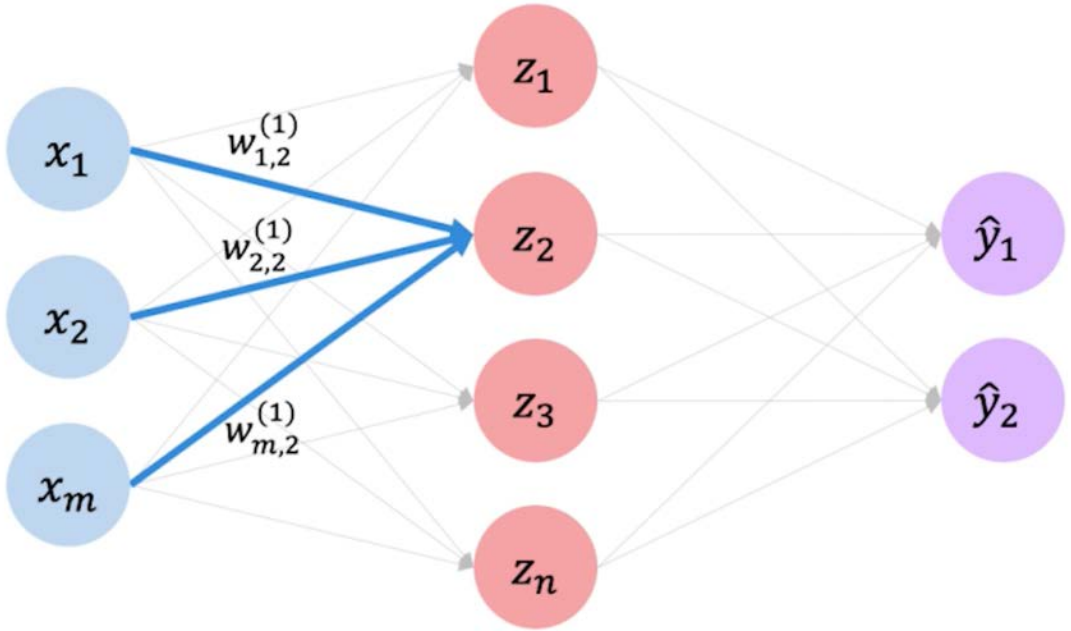


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)}$$

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

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Red Neuronal con una solo capa oculta

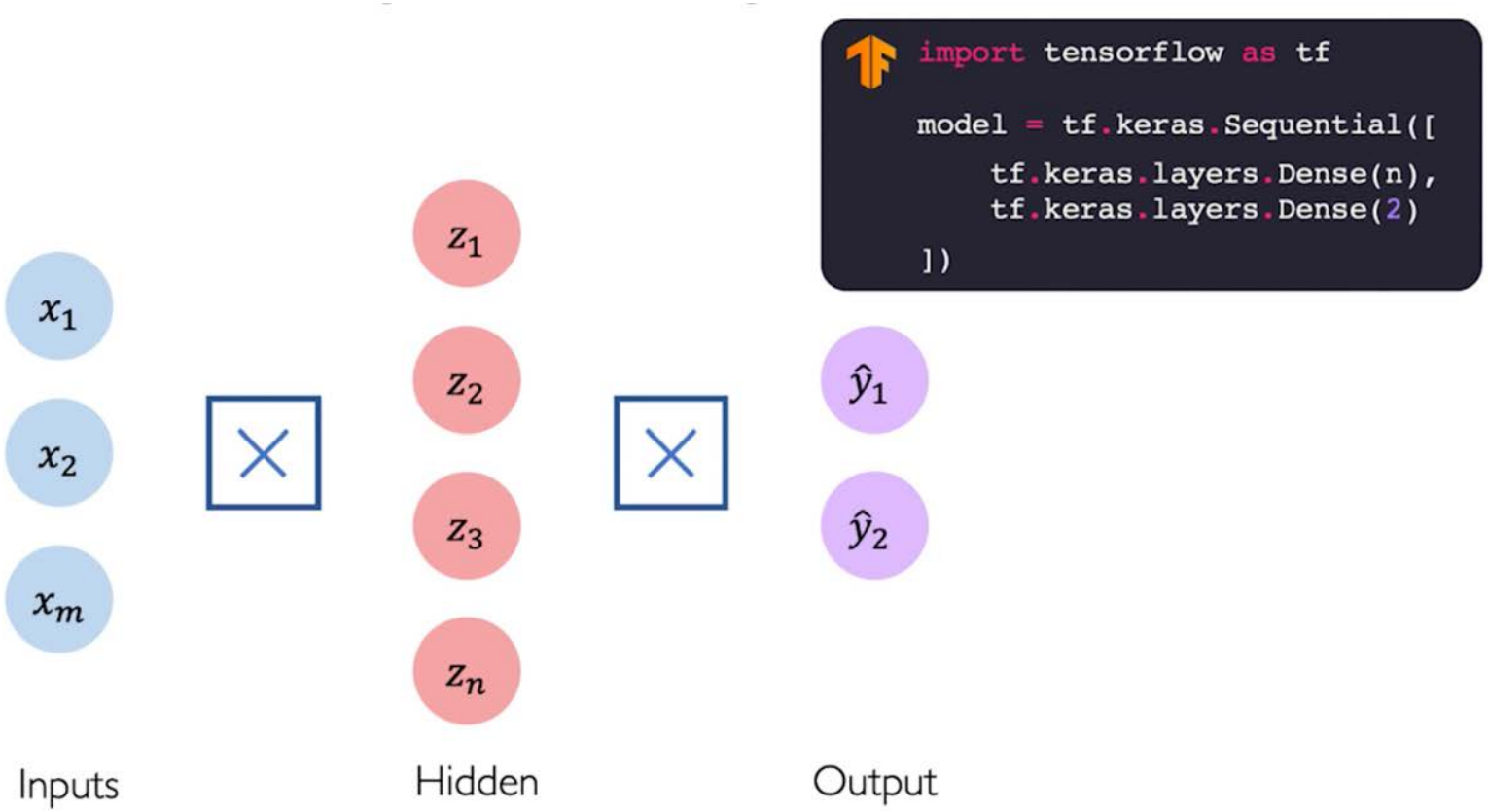


$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,2}^{(1)}$$

$$= w_{0,2}^{(1)} + x_1 \, w_{1,2}^{(1)} + x_2 \, w_{2,2}^{(1)} + x_m \, w_{m,2}^{(1)}$$

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

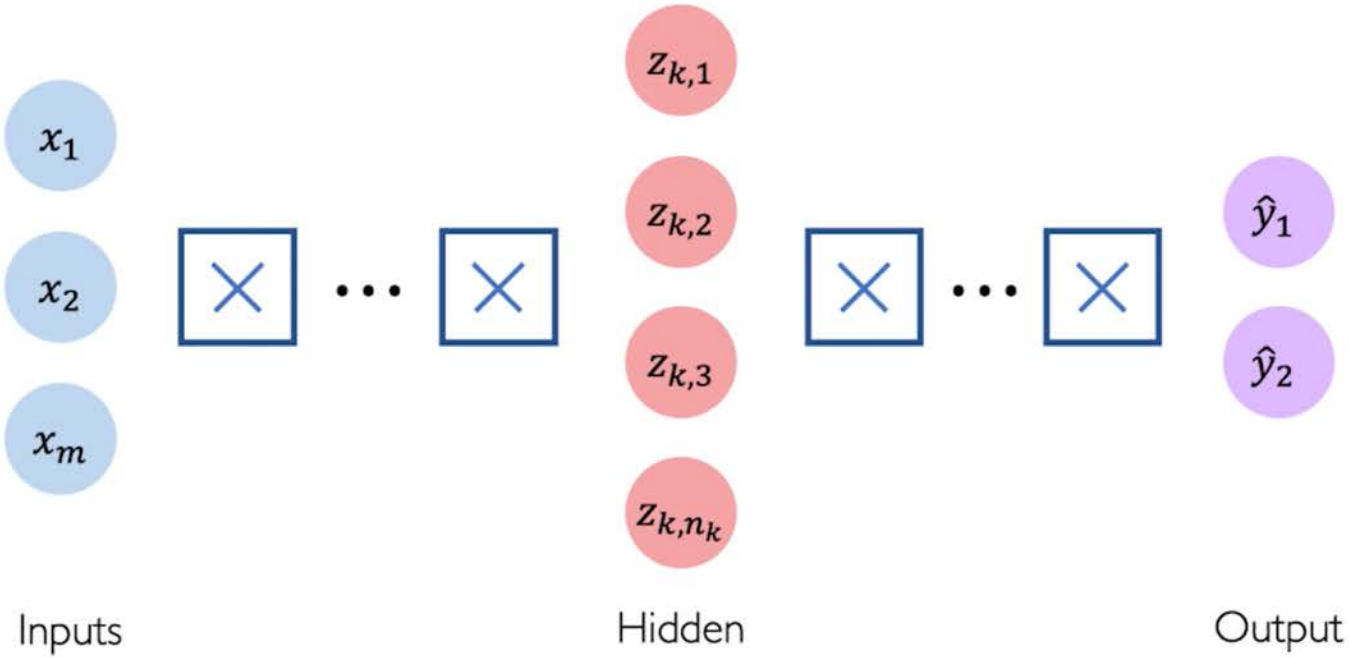
Red Neuronal con una solo capa oculta




```python
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(n),
    tf.keras.layers.Dense(2)
])
```

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Red Neuronal Profunda




```tensorflow
import tensorflow as tf

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

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

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Problema de Aplicación

Redes Neuronales

Problema de aplicación

Es posible predecir si un estudiante pasará o no un curso:

Se asumirá un modelo muy simple con dos características

$$x1 = Número\ de\ clases\ a\ las\ que\ asistió$$

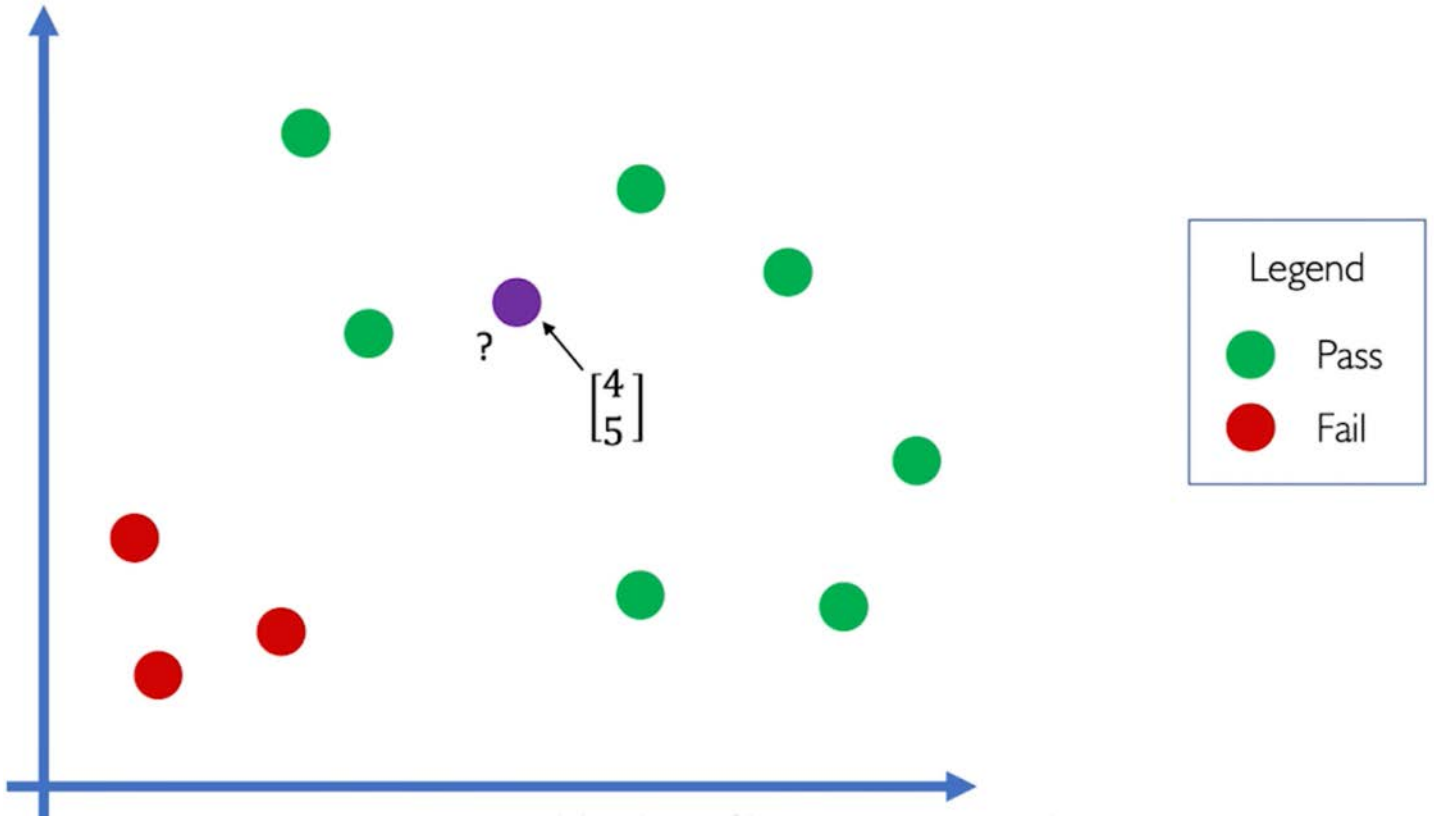
$$x2 = Horas\ que\ dedicó\ a\ estudiar\ por\ su\ cuenta$$

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Problema de aplicación

$x1 = Número\ de\ clases$

$x2 = Horas\ \ de\ estudio$

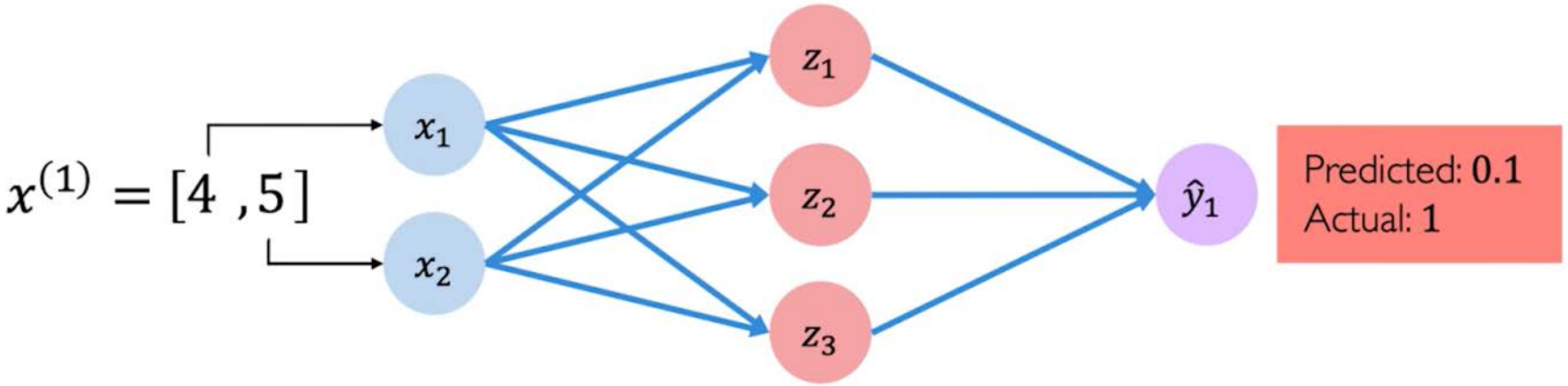


Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Cuantificando pérdida

La **pérdida** (loss) en una red, mide el costo de hacer malas predicciones.

- Se trata de reducir la pérdida al máximo
- Perdida pequeña => Valor predecido es muy parecido al real.



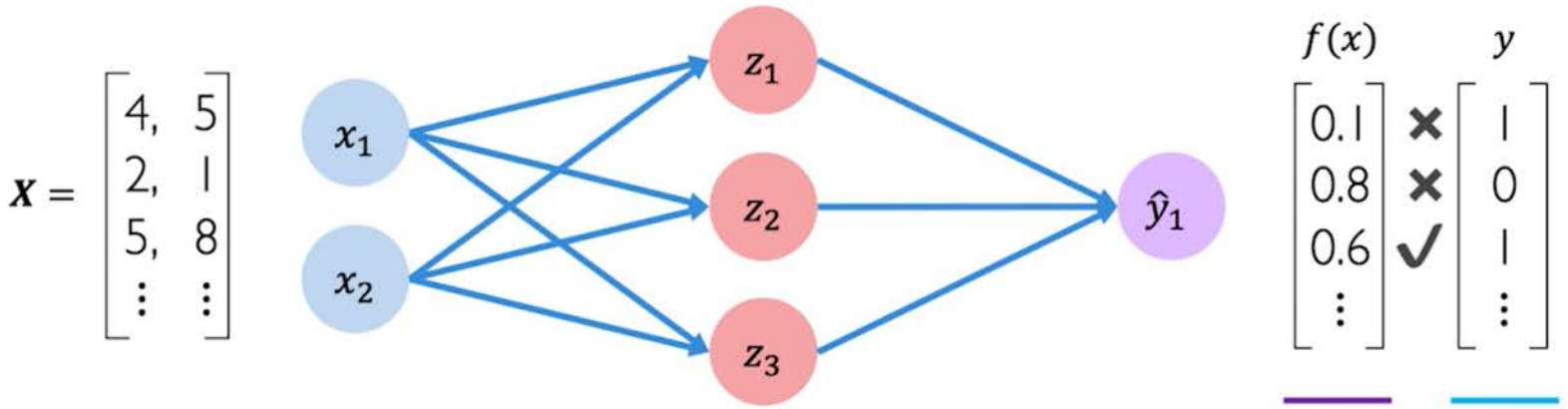
$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Función de Pérdida Empírica

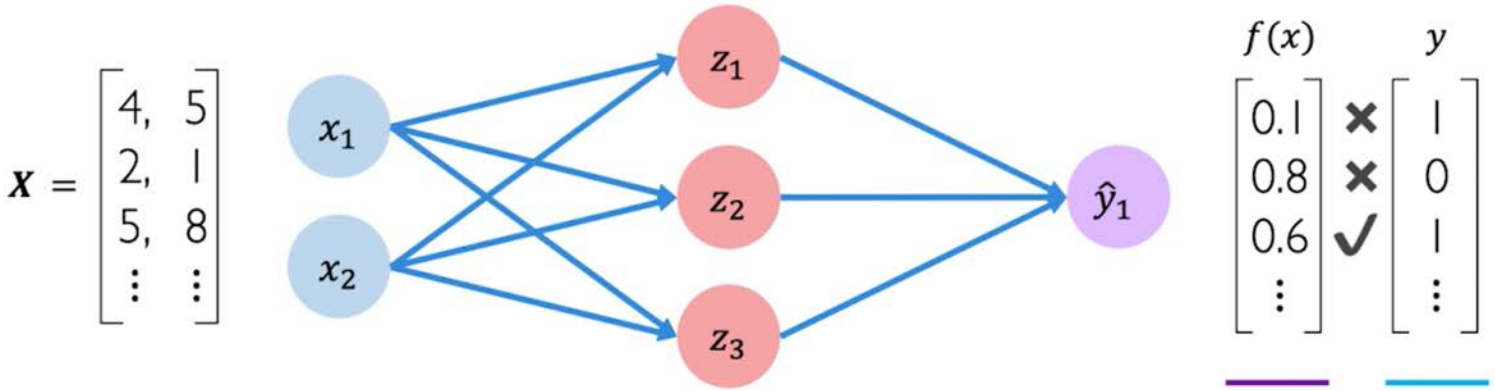
La **pérdida empírica** (empirical loss) mide la pérdida total sobre un dataset completo.



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Pérdida de entropía cruzada binaria (Binary cross entropy loss)

La **pérdida de entropía cruzada**, se puede utilizar en modelos que obtienen como salida una decisión binaria 0 / 1 (Si / No).



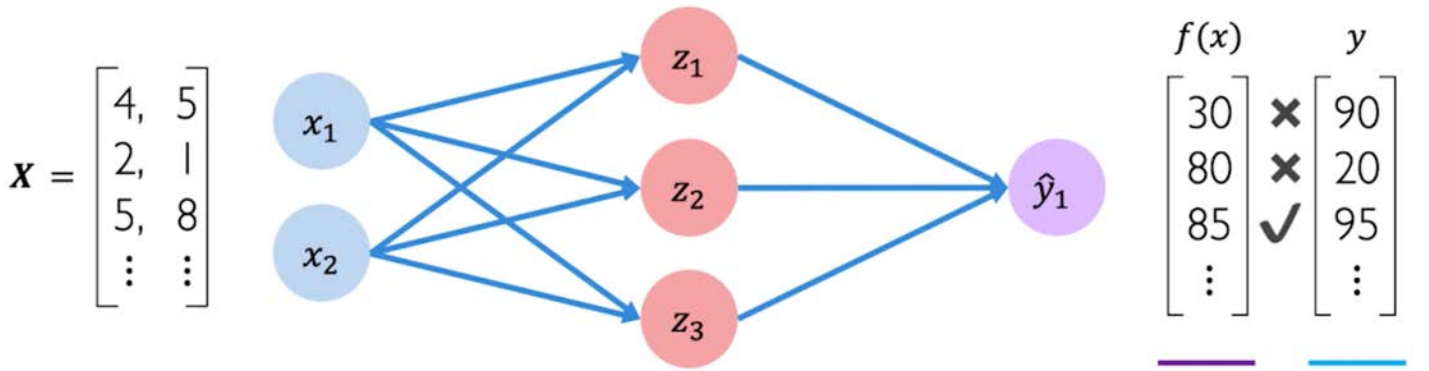
$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \underbrace{\log\left(f\left(x^{(i)}; W\right)\right)}_{\text{Predicted}} + \underbrace{(1 - y^{(i)})}_{\text{Actual}} \underbrace{\log\left(1 - f\left(x^{(i)}; W\right)\right)}_{\text{Predicted}}$$

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

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Pérdida de Error Cuadrático Medio (Mean square error loss)

Este tipo de pérdida se puede emplear con modelos de regresión que obtienen como salida números reales continuos.




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

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Deep Learning Redes Neuronales

Proceso de Entrenamiento

Optimización de Pérdida

Se requiere encontrar los pesos para obtener una pérdida mínima

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

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

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Optimización de Pérdida

Se requiere encontrar los pesos para obtener una pérdida mínima

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

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

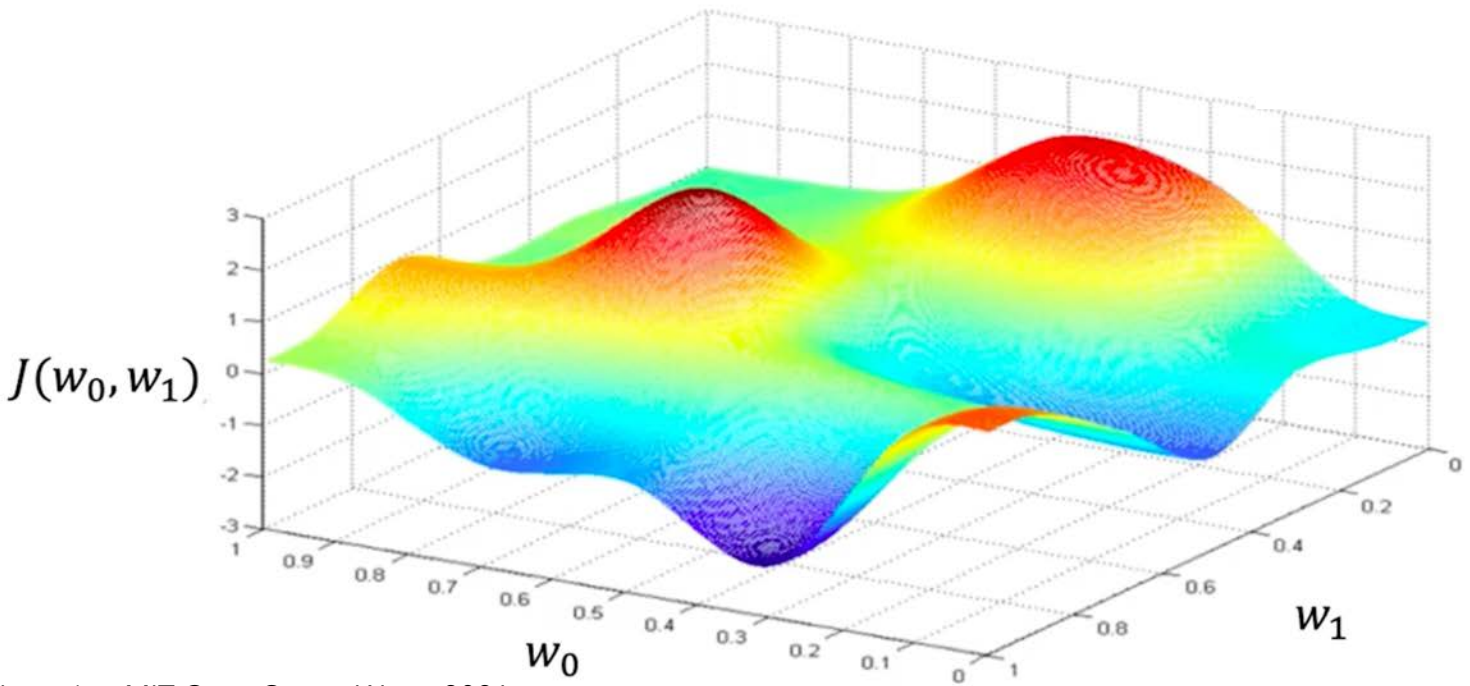
$$\boldsymbol{W} = \{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \dots\}$$

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Optimización de Pérdida

Pérdida: Función que representa los pesos de la red

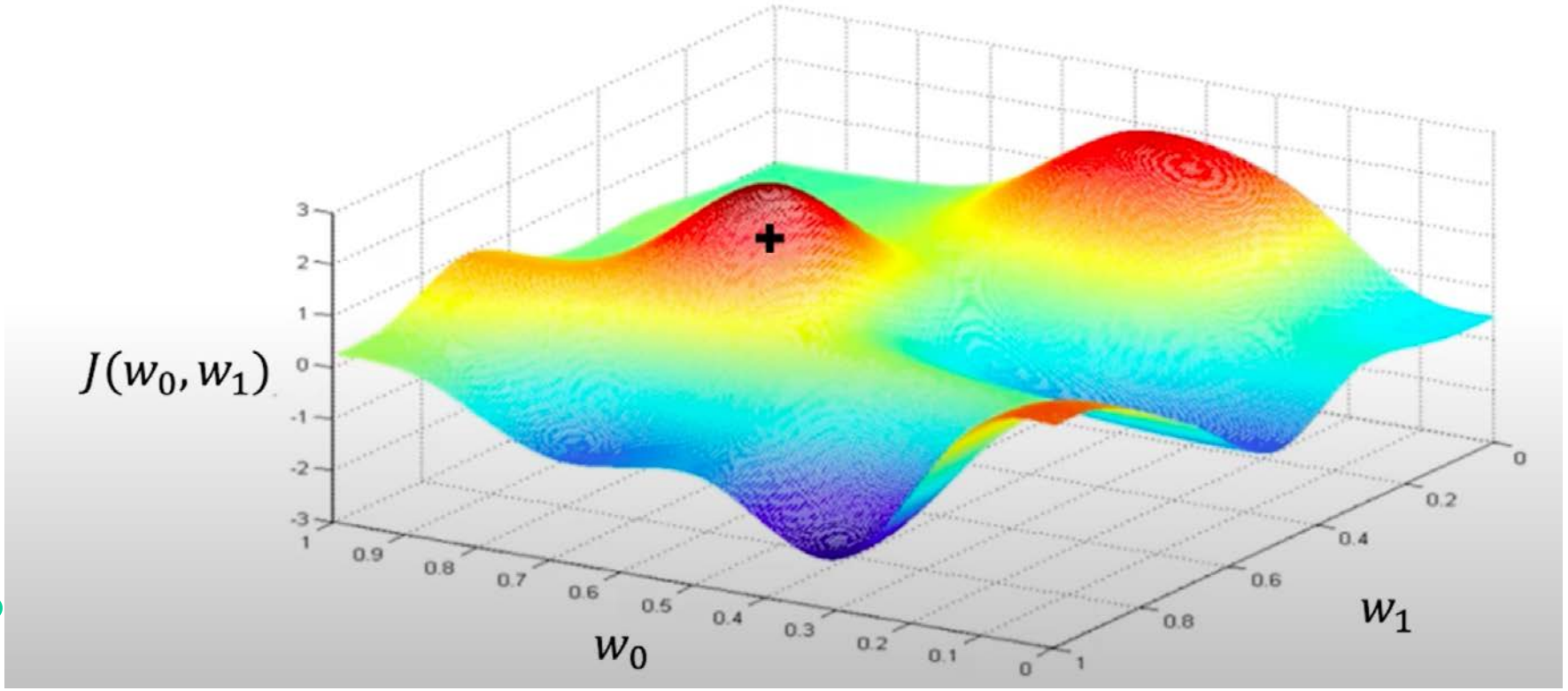
$$W^* = \underset{W}{\operatorname{argmin}} J(W)$$



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Optimización de Pérdida

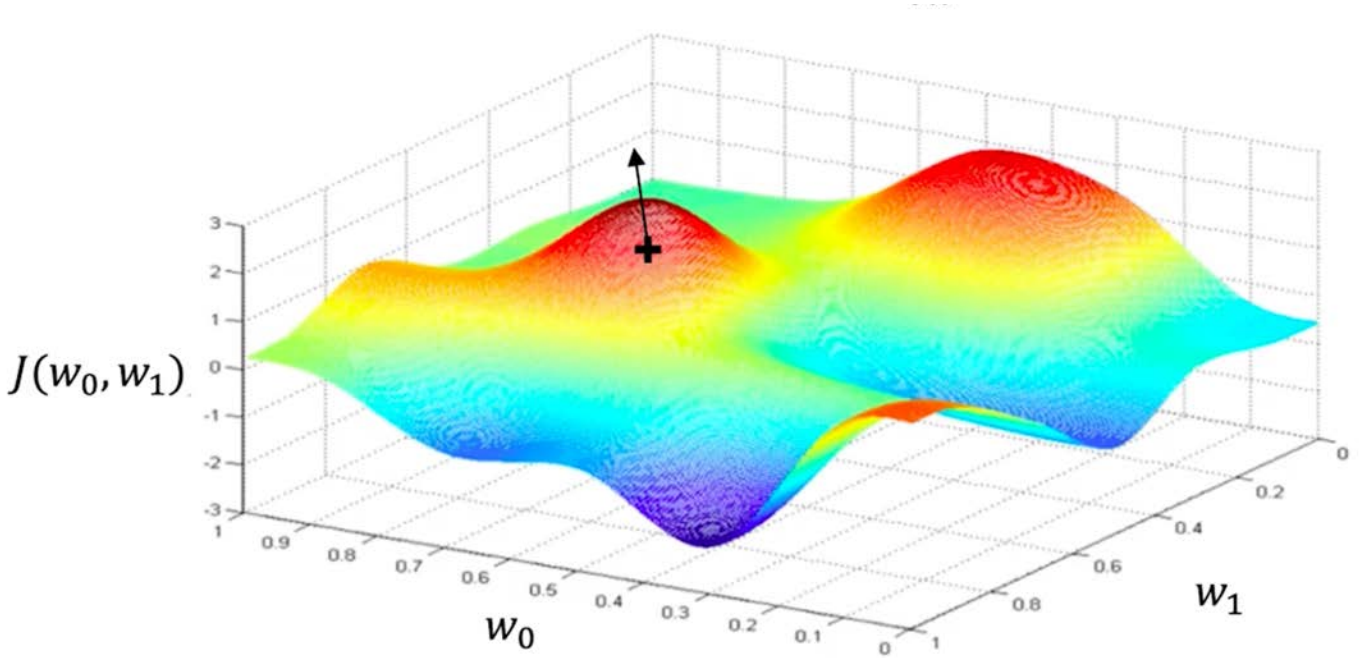
Se inicializa un punto aleatorio (w_0, w_1)



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Optimización de Pérdida

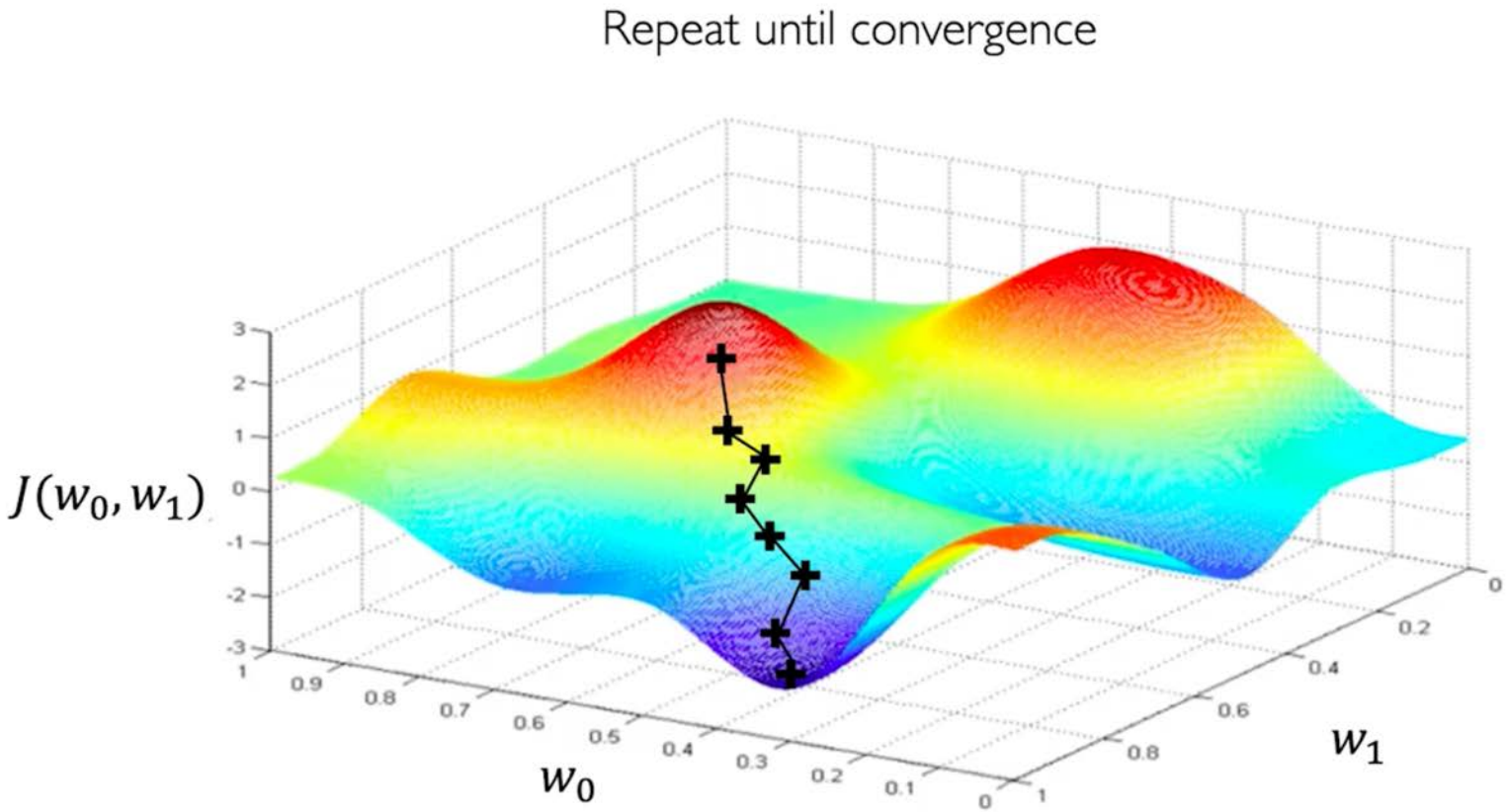
Se calcula el gradiente $\frac{\partial J(W)}{\partial W}$



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Descenso de gradiente (Gradient descent)

Se repite hasta que converja a un mínimo



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Descenso de gradiente (Gradient descent)

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

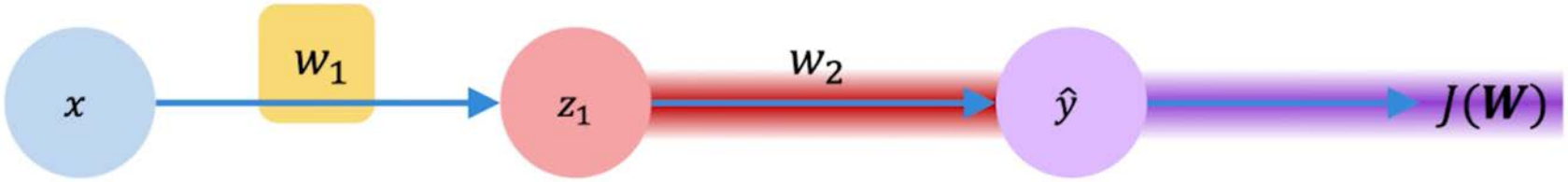
```python
import tensorflow as tf

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
```

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Back Propagation

Neural Networks

Calculando gradientes: Back Propagation



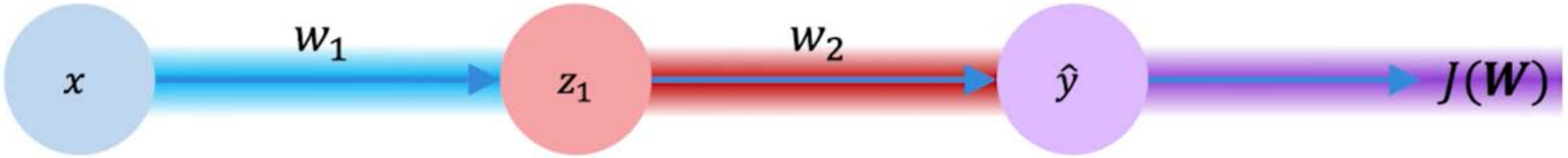
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Calculando gradientes: Back Propagation

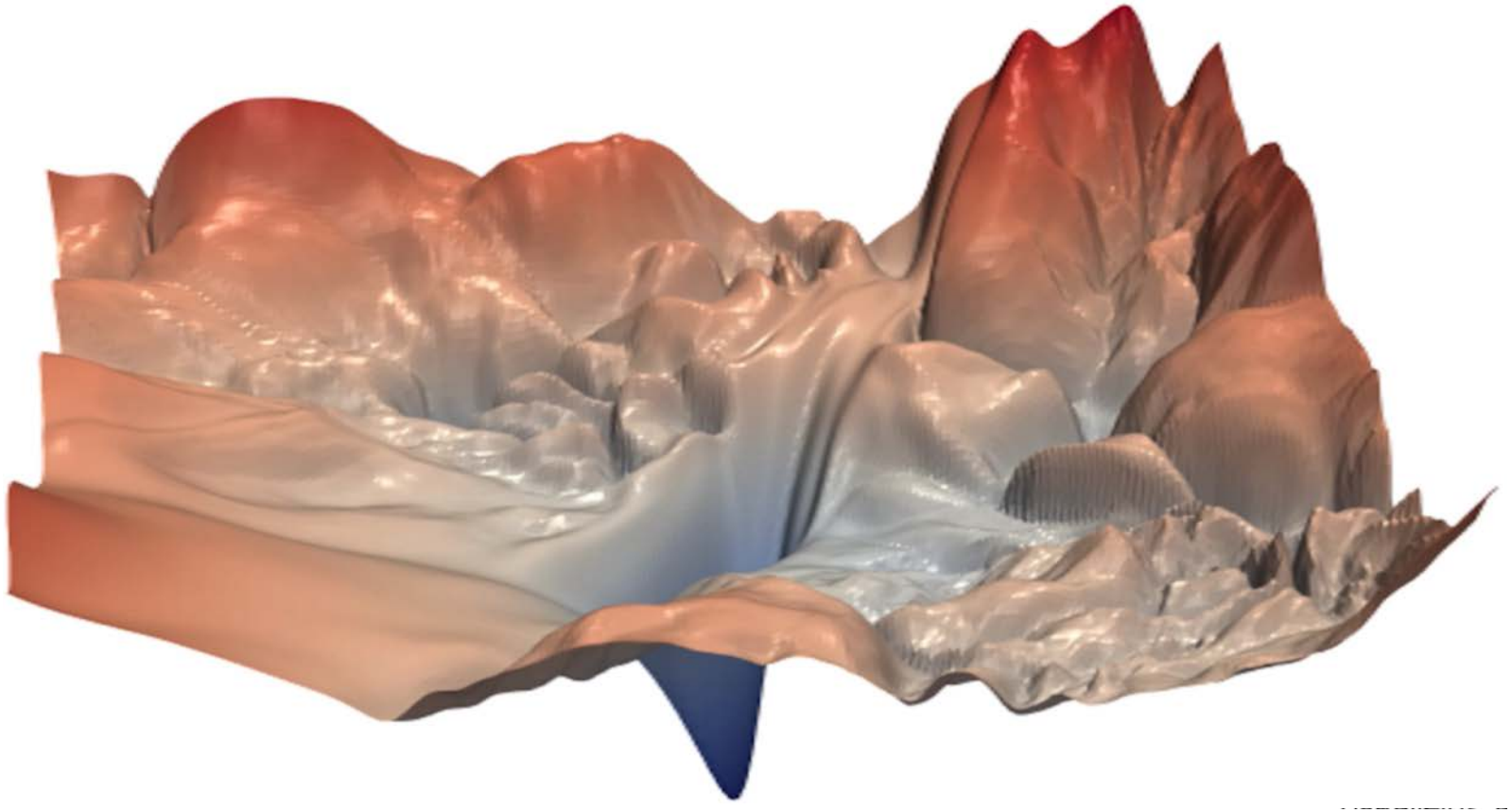


$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Se repite para cada peso en la red usando los gradientes de las capas siguientes

Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Entrenamiento de una Red Neuronal en la Práctica



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

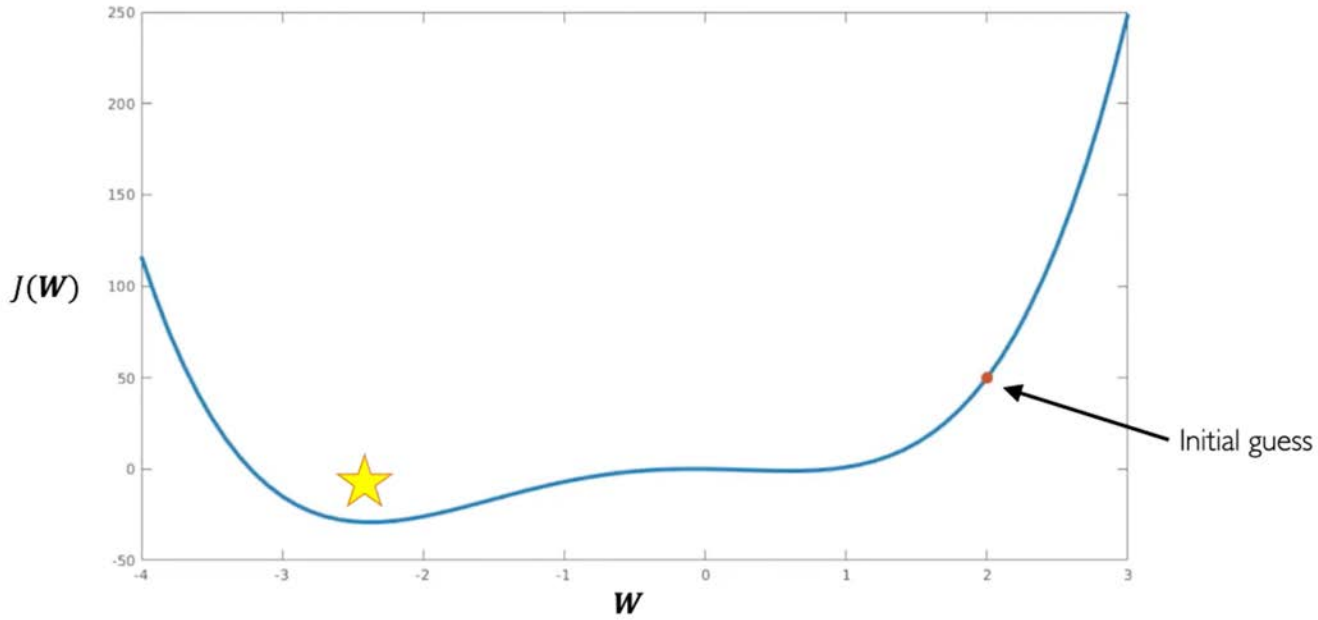
"Visualizing the loss landscape of neural nets". Dec 2017.

Función de pérdida puede ser difícil de optimizar

Optimización a través de método del gradiente

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

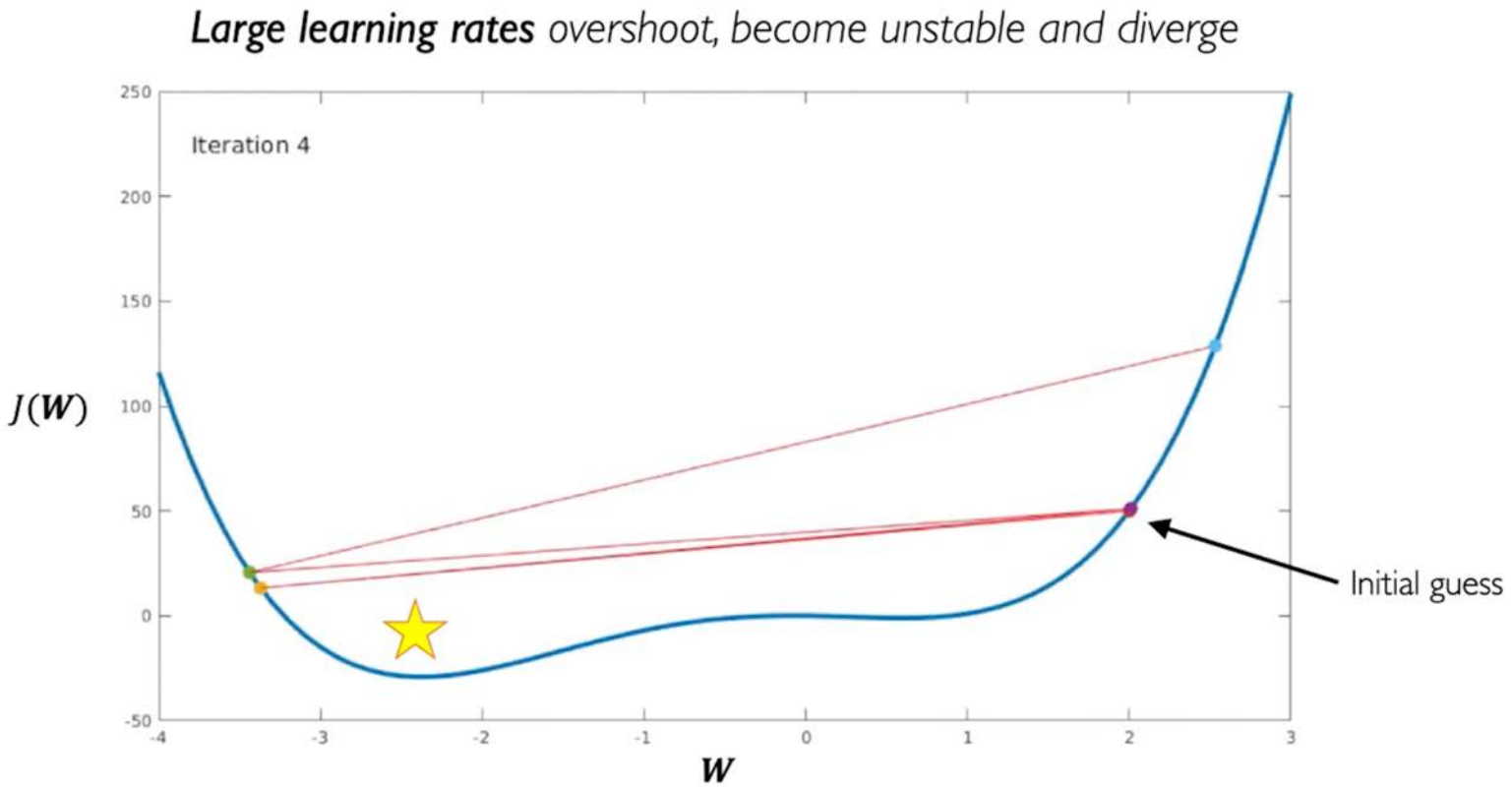
Tasa de aprendizaje
(learning rate)



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Función de pérdida puede ser difícil de optimizar

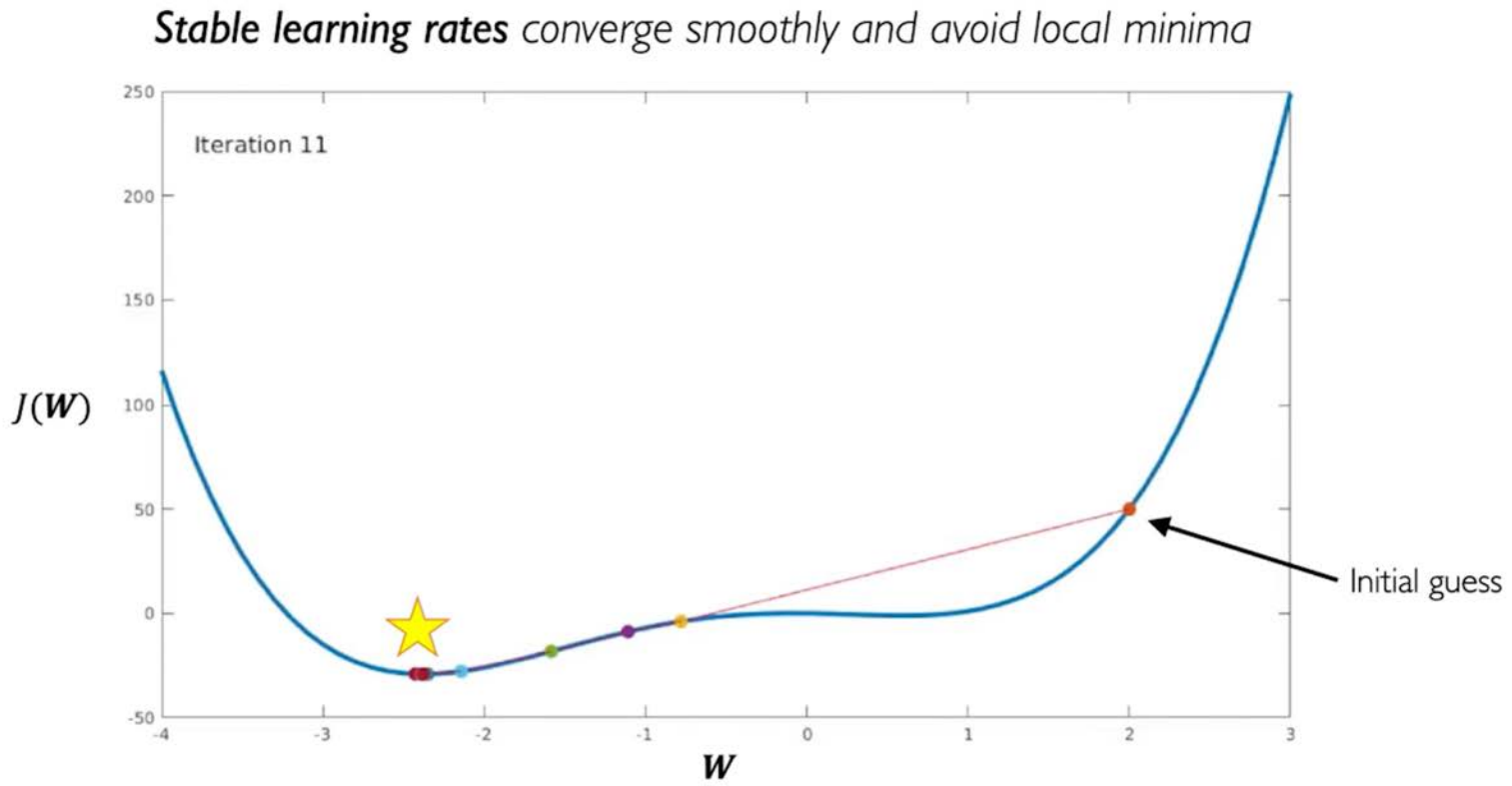
Optimización a través de método del gradiente



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Función de pérdida puede ser difícil de optimizar

Optimización a través de método del gradiente



Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Algoritmos de Gradient Descent

Algorithm	TF Implementation	Reference
• SGD	`tf.keras.optimizers.SGD`	Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.
• Adam	`tf.keras.optimizers.Adam`	Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.
• Adadelta	`tf.keras.optimizers.Adadelta`	Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.
• Adagrad	`tf.keras.optimizers.Adagrad`	Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.
• RMSProp	`tf.keras.optimizers.RMSProp`	

Diseñando una primer aplicación con Tensor Flow

```python
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))
```

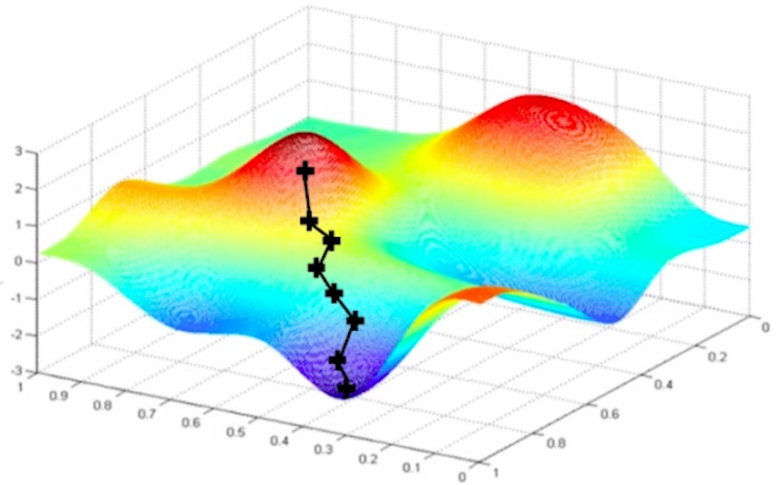
Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Redes Neuronales en la práctica: Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

Can be very **computationally intensive** to compute!

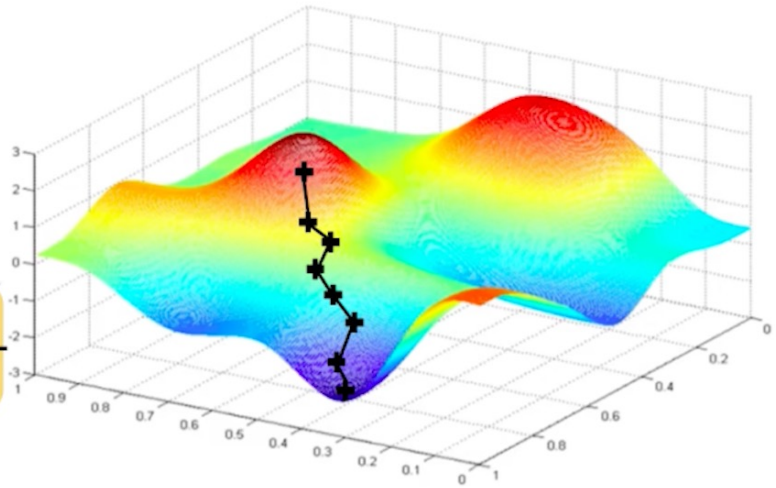


Si se calcula sobre cada punto es computacionalmente muy caro.

Redes Neuronales en la práctica: Stochastic Gradient Descent (Mini-batches)

Algorithm

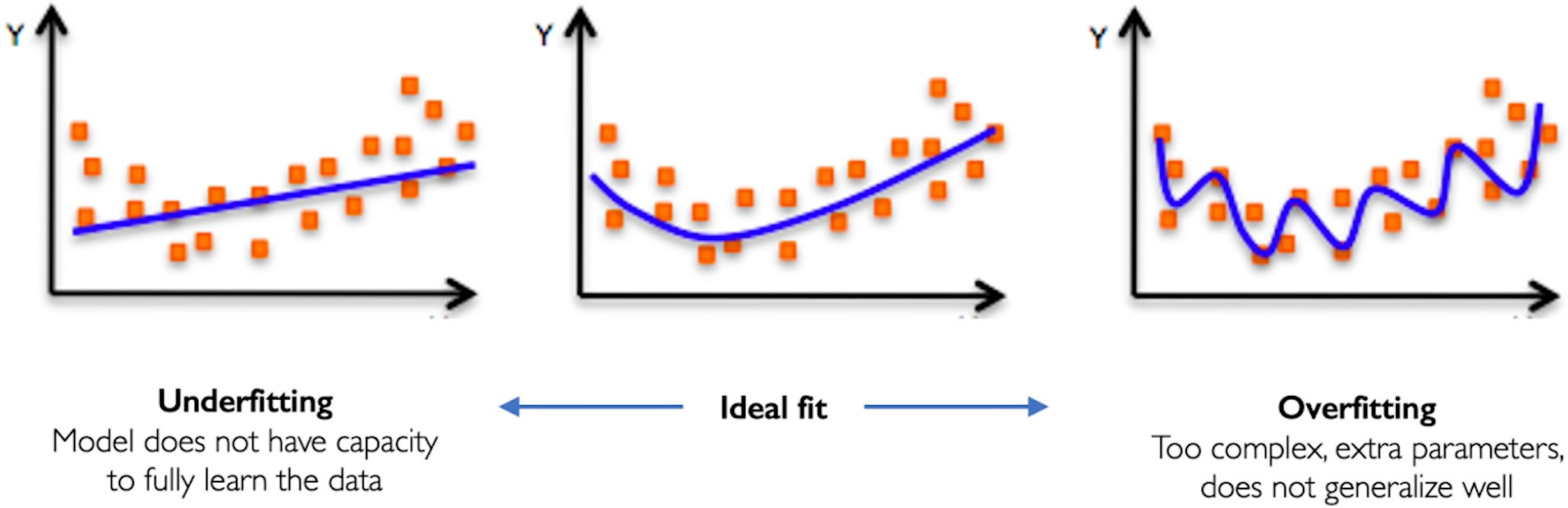
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



Fast to compute and a much better estimate of the true gradient!

Mini batches -> Son del orden de cientos o miles de elementos.

El problema de Sobre-estimaciones

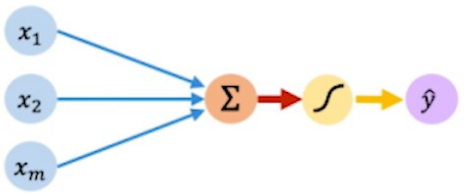


Fuente: Introduction to Deep Learning MIT OpenCourseWare, 2021

Resumen

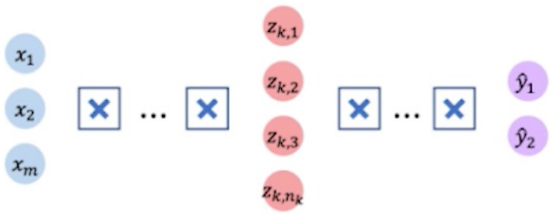
The Perceptron

- Structural building blocks
- Nonlinear activation functions



Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization