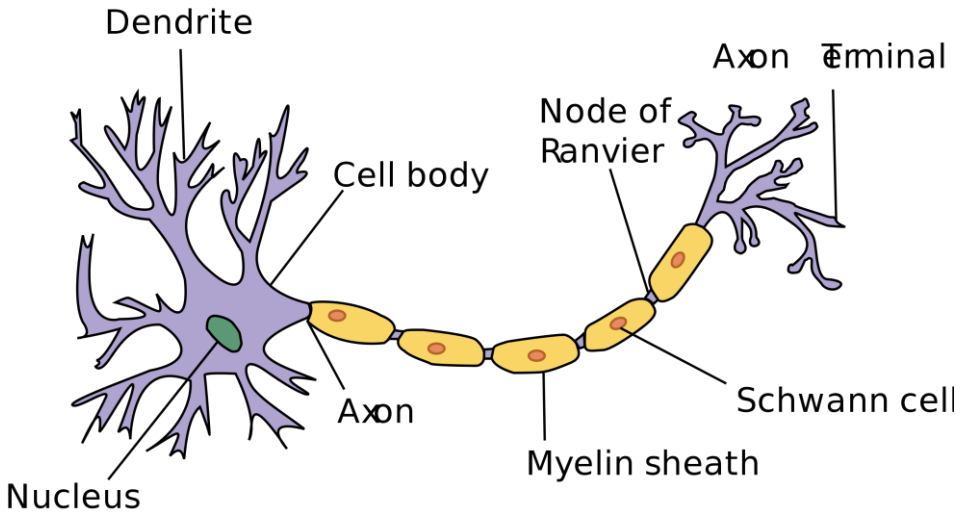
Redes Neuronales

Perceptrón, Funciones de Activación y Descenso por el Gradiente

Agenda

1. Perceptrón vs Neurona Biológica
2. Activación del perceptrón
    1. Sigmoide
    2. Tanh
    3. Relu
3. Funciones de pérdida
4. Aprendizaje por Gradiente
    1. Intuición del Aprendizaje por el Gradiente
    2. Hiperparámetro Learning Rate
    3. Mini-batches
5. Introducción a Tensorflow y Keras.



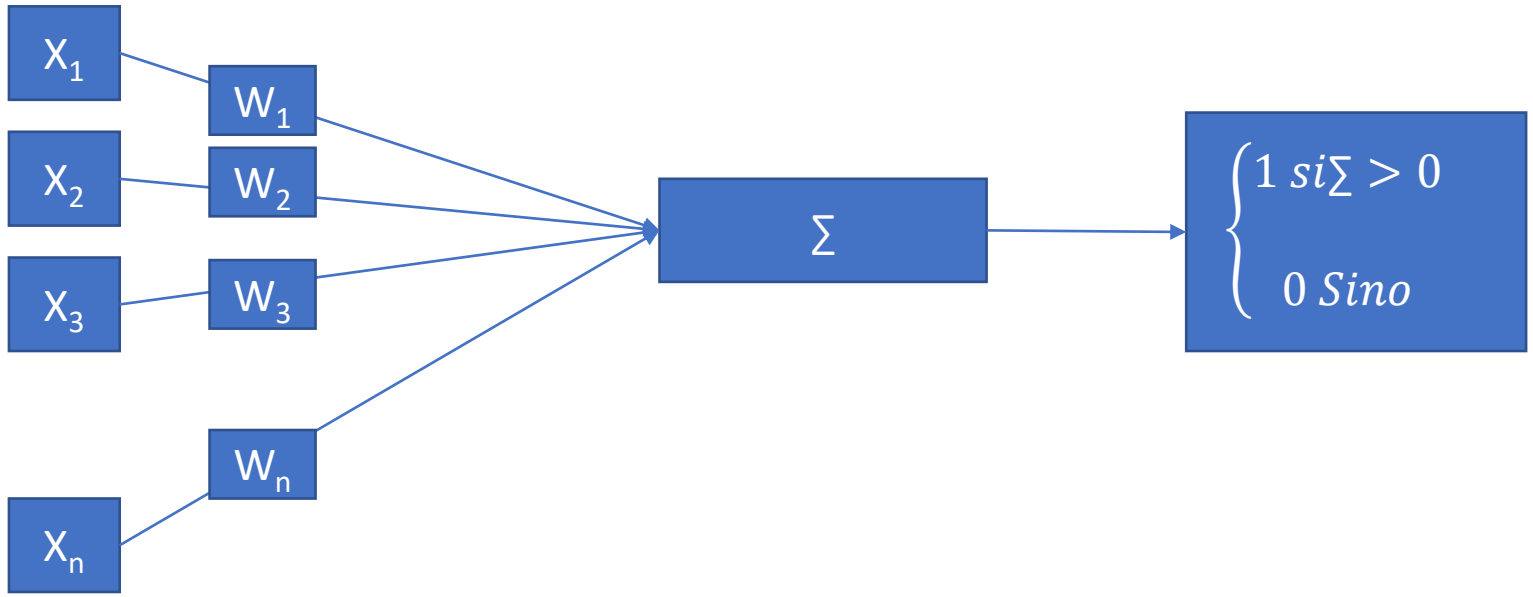
"Anatomy and Physiology" by the US National Cancer Institute's Surveillance, Epidemiology and End Results (SEER) Program .

TensorFlow

Perceptrón

Original

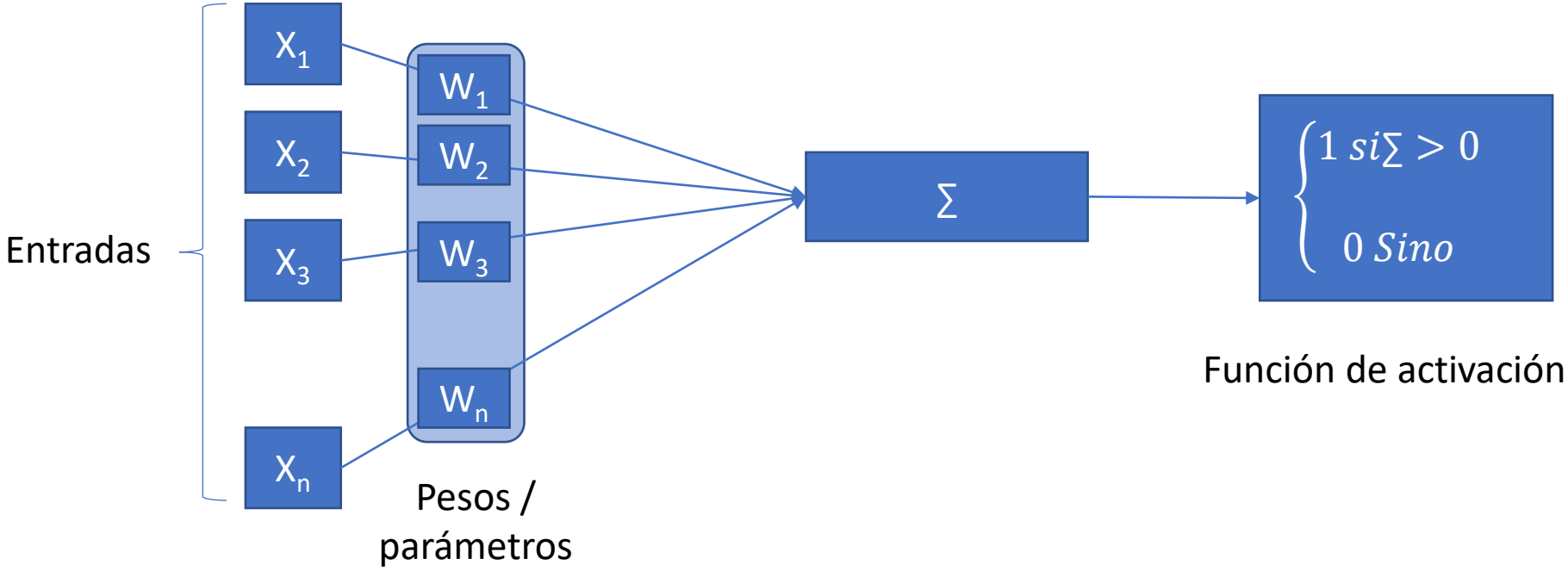
- Originalmente introducido en 1958 por Frank Rosenblatt.
- Utilizado para clasificación binaria
- Se basaba en el concepto de un neurona que recibe estímulos y se dispara o no dependiendo de su estado interno.



Perceptrón

Original

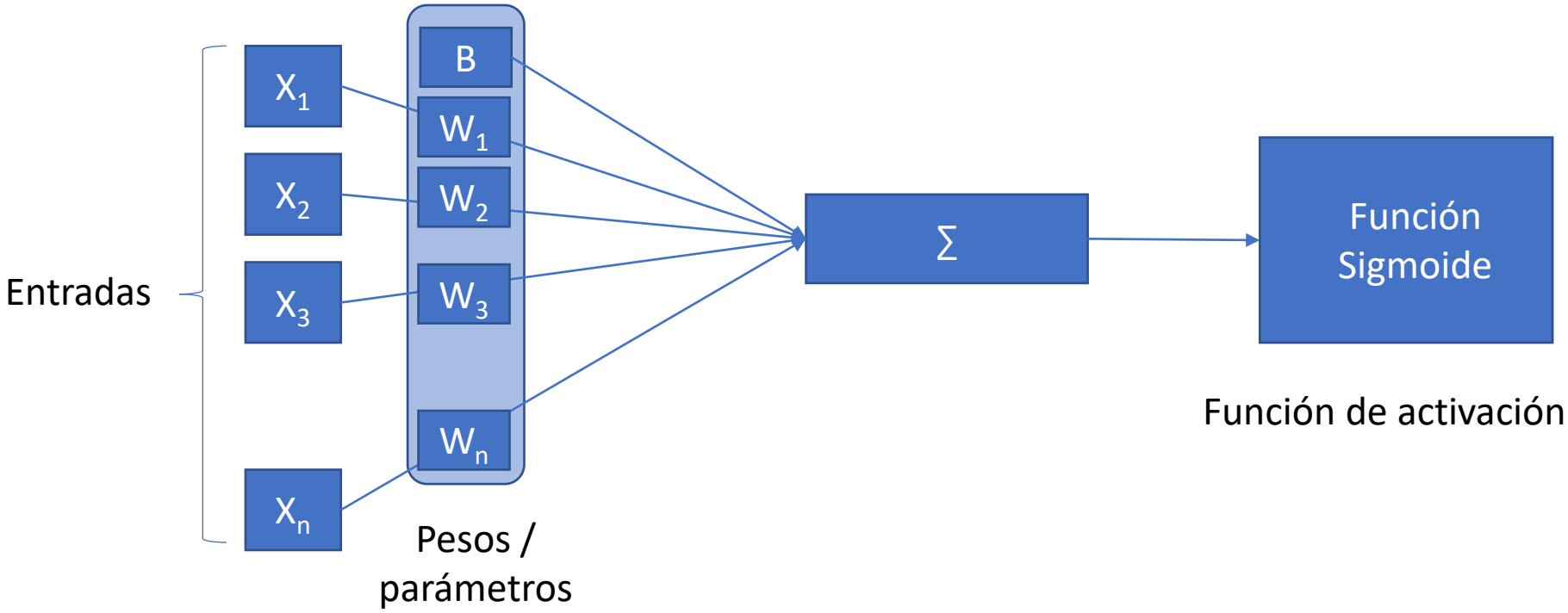
- Originalmente introducido en 1958 por Frank Rosenblatt.
- Utilizado para clasificación binaria
- Se basaba en el concepto de un neurona que recibe estímulos y se dispara o no dependiendo de su estado interno.



Perceptrón

Regresión logística

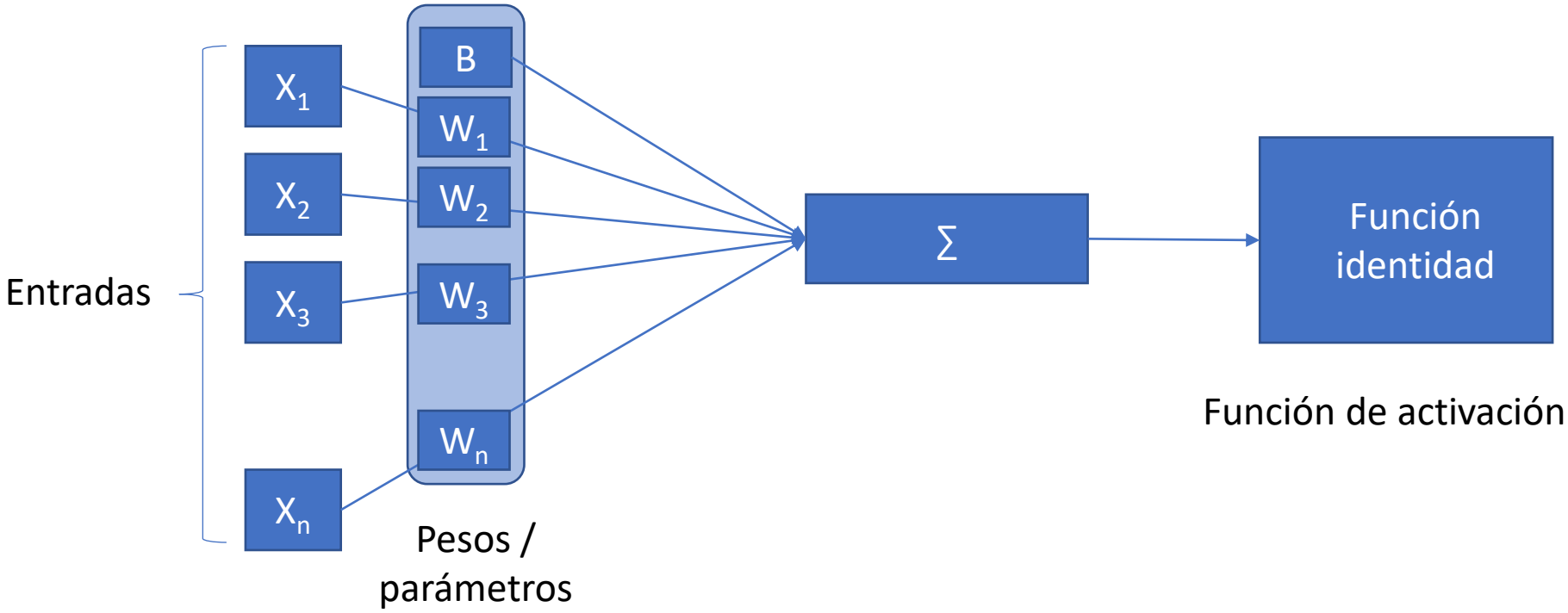
- Agregando un parámetro más conocido como *bias* y cambiando la función de activación se puede obtener la regresión logística.
- Se puede pensar como una aproximación al perceptrón original.
- Es más fácil de entrenar.



Perceptrón

Regresión lineal

- Cambiando la función sigmoide por la identidad, se puede llegar a una regresión lineal



Perceptrón

Entrenando una regresión lineal

- Considerando una regresión lineal, vamos a entrenar los parámetros.
- La regresión lineal se define como:

$$H(Y, W, b) = x_1 \cdot w_1 + x_2 \cdot w_2 + \cdots + x_n \cdot wn + b = x \cdot w + b$$

- X es nuestro matriz de instancias por características. W y b son los parámetros de nuestra regresión que tenemos que entrenar.
- Para entrenarlo, debemos definir una función de pérdida (*loss*). En la regresión lineal se suele usar el error cuadrático medio:

$$loss(Y, \hat{Y}) = L(Y, \hat{Y}) = \frac{\sum(y_i - \hat{y_i})^2}{n}$$

- Y son nuestro valores asociados a Y, $\hat{Y}$ son las predicciones y n es la cantidad de muestras.

Entrenando una regresión lineal

El objetivo es

$$\underset{W,b}{\operatorname{argmin}}\, loss(Y, H(X, W, b))$$

Es decir, la idea es buscar los valores de w y b que minimicen la función de pérdida aplicada sobre los valores esperados y la salida de la regresión lineal.

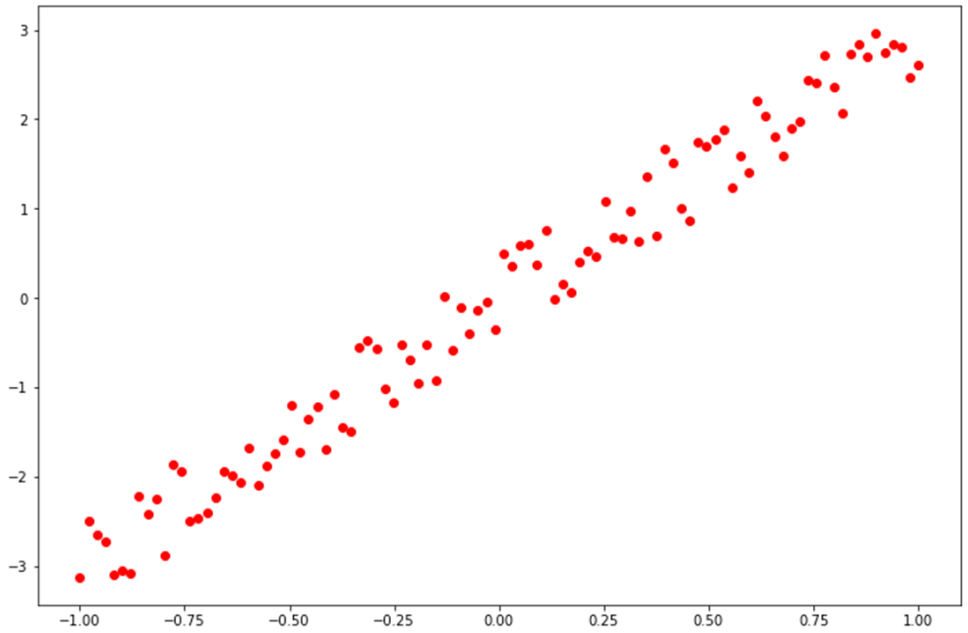
Pero… ¿Cómo podemos encontrar esos valores?

Aprendizaje por gradiente

Entrenando una regresión lineal

- Por simplicidad vamos a considerar el siguiente conjunto de datos artificial:
  - Vamos a generar 100 puntos de datos
  - x será un escalar en vez de un vector
  - Se agregará un poco de ruido

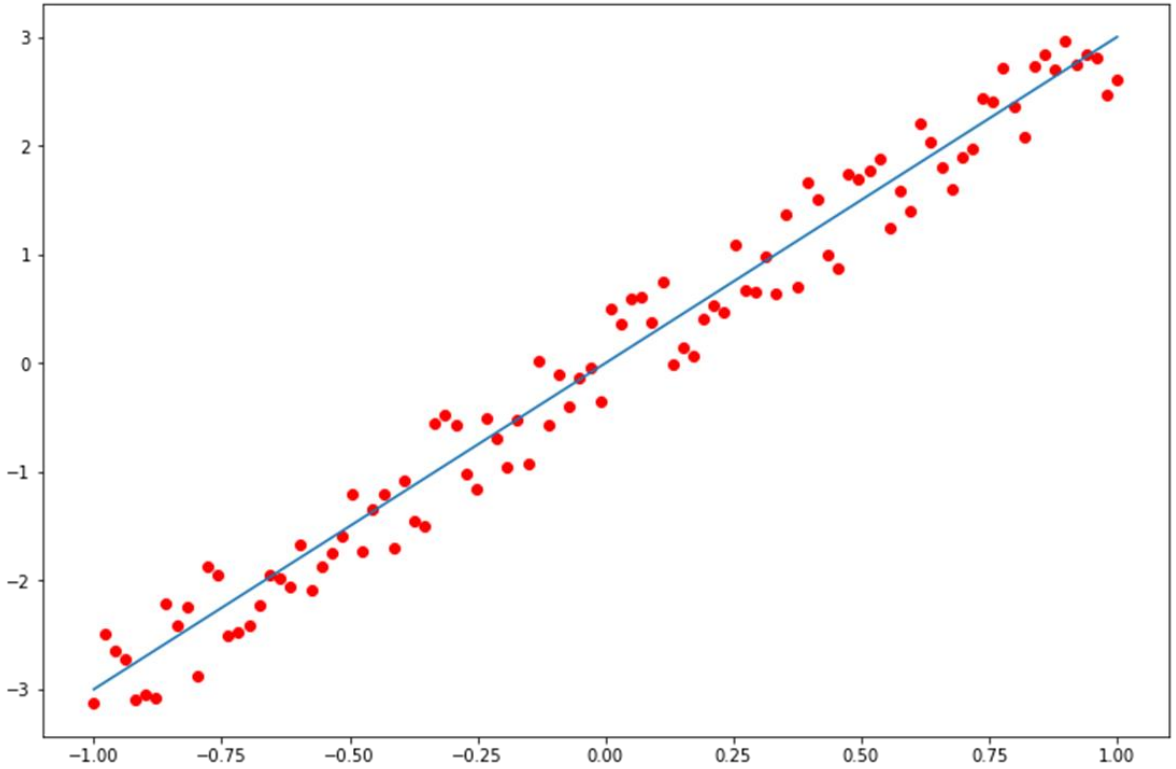
$$y = x + 3 + rand(-0{,}5, ; 0{,}5)$$



Aprendizaje por gradiente

Entrenando una regresión lineal

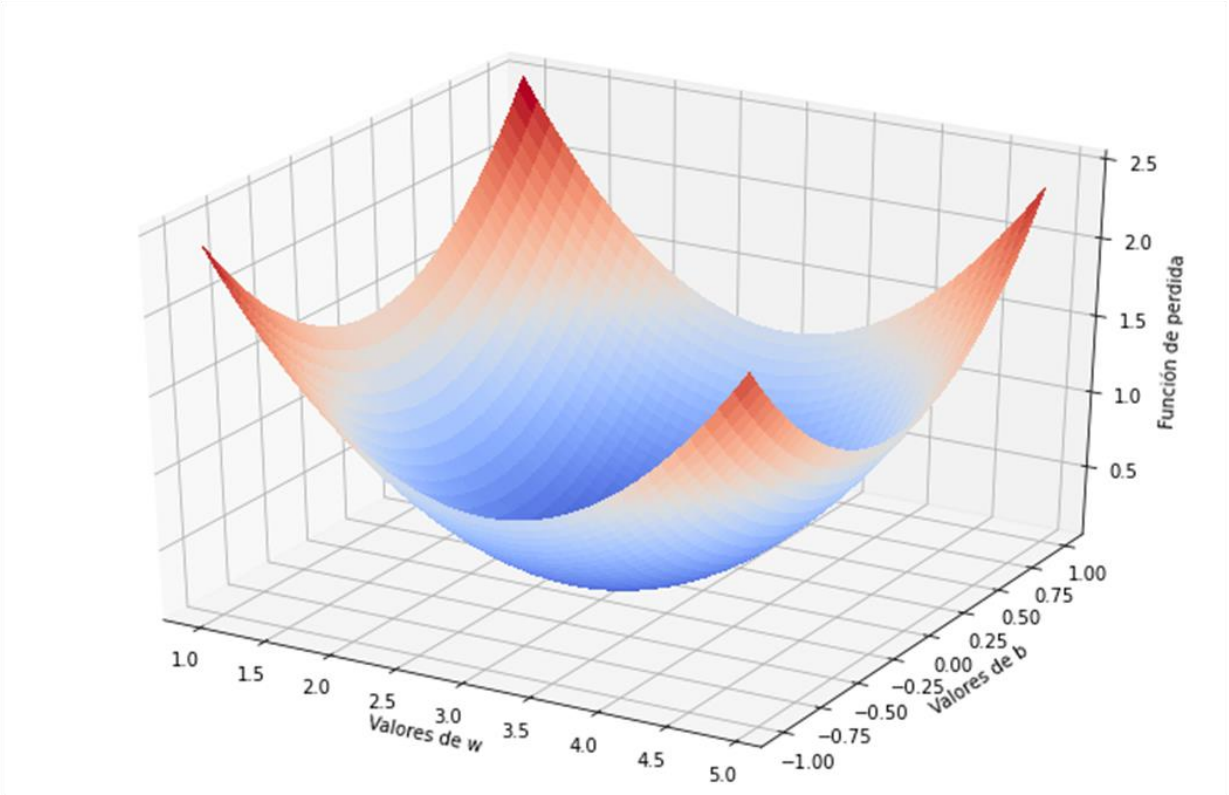
- Obviamente, si aplicamos una regresión lineal con $w = 3$ y $b = 0$ tenemos una recta que ajusta los datos con una perdida cercana a la mínima.
- El MSE para esta ejecución es 0.08030984087067287.



Aprendizaje por gradiente

Entrenando una regresión lineal

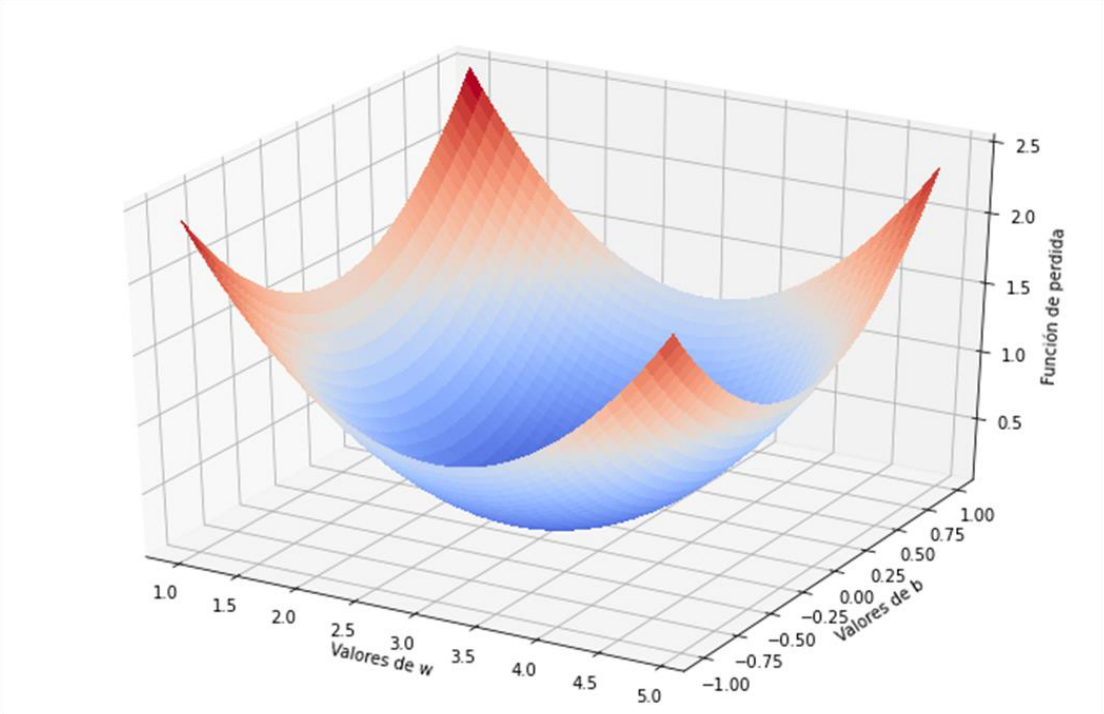
Para buscar los valores de w y b podríamos probar distintos valores y quedarnos con el que minimice el valor de perdida. Pero…



Aprendizaje por gradiente

Entrenando una regresión lineal

- Cada prueba requiere calcula la perdida sobre todos los elementos en el conjunto de entrenamiento.
- Problemas de escalabilidad con el número de características e instancias de entrenamiento.
- Imposible evaluar sobre todos los posibles valores de w y b.
- Por lo que se necesita una mejor estrategia para buscar los valores de los parámetros.

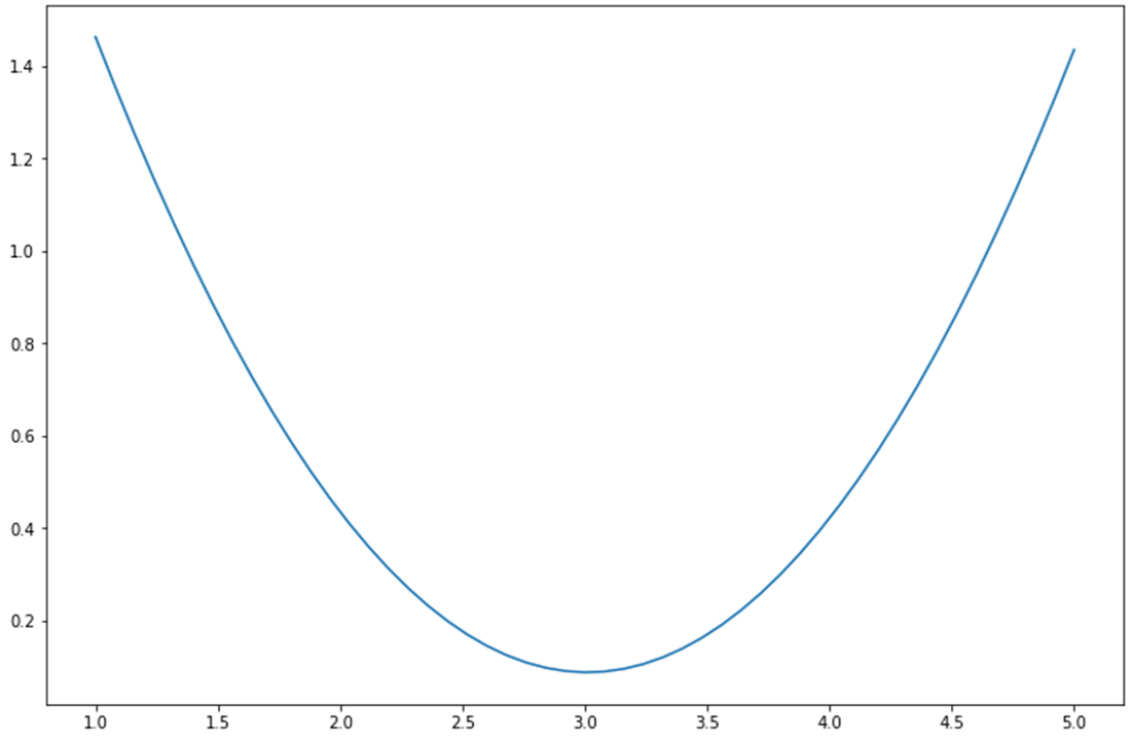


Aprendizaje por gradiente

Entrenando una regresión lineal

- Para simplificar el análisis, fijamos $b = 0$.
- Por lo tanto, podemos graficar la perdida en función de w
- Considerando Y, X y b son constantes:

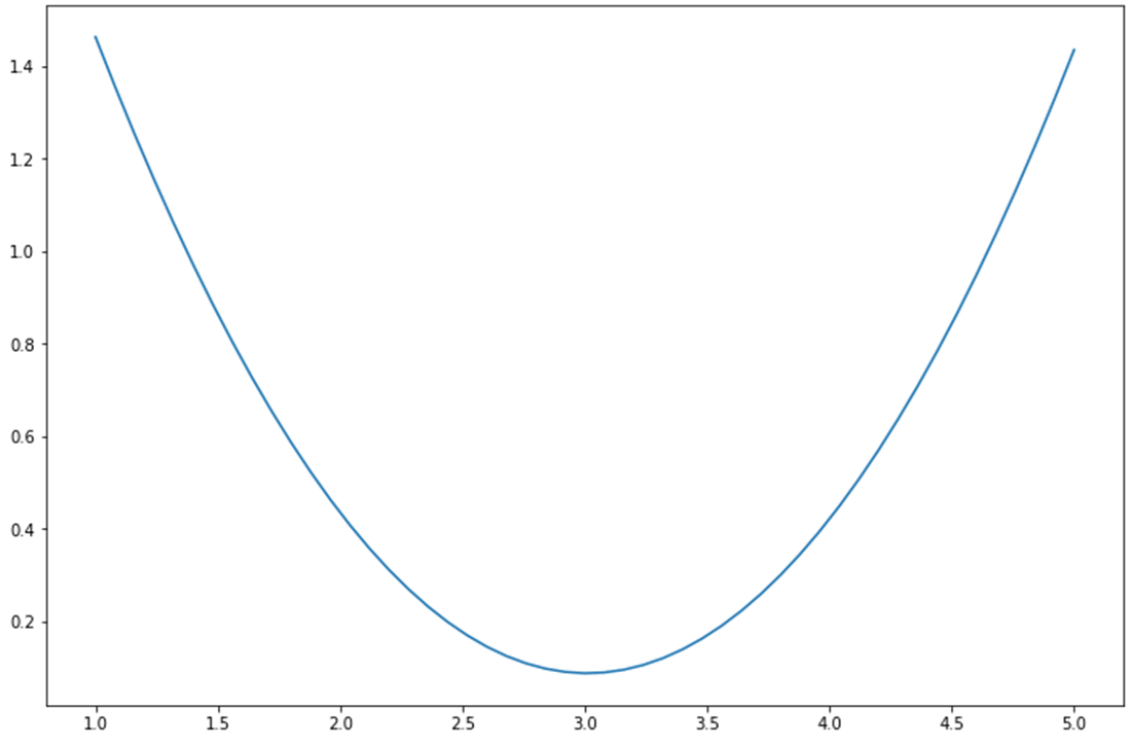
$$L\left(Y, \hat{Y}\right) = L\left(Y, H(X, w, b)\right) = L(w)$$



Aprendizaje por gradiente

Entrenando una regresión lineal

- Considerado esta información, se podrían probar 2 valores cercanos para w y determinar como varia la función de perdida en la localidad.

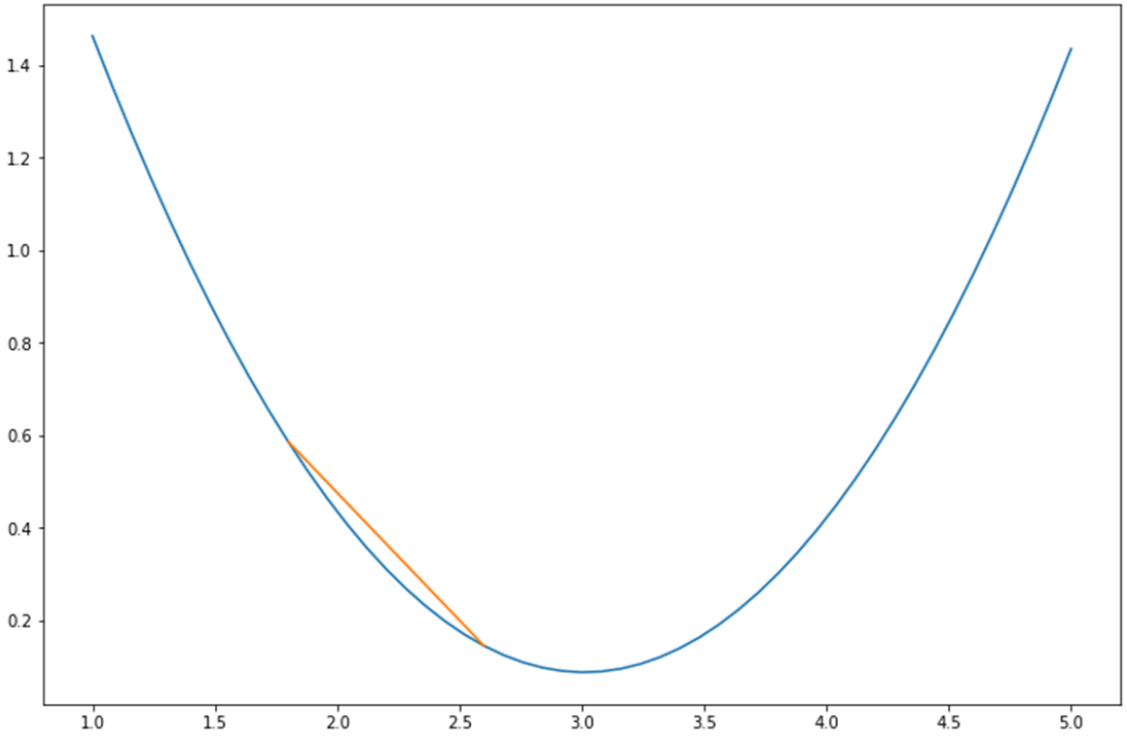


Aprendizaje por gradiente

Entrenando una regresión lineal

En el ejemplo, vemos como se puede trazar una recta entre dos perdidas dadas. La pendiente de esta recta puede ser calculada de la siguiente forma.

$$pendiente(w, \Delta) = \frac{L(w) - L(w + \Delta)}{\Delta}$$

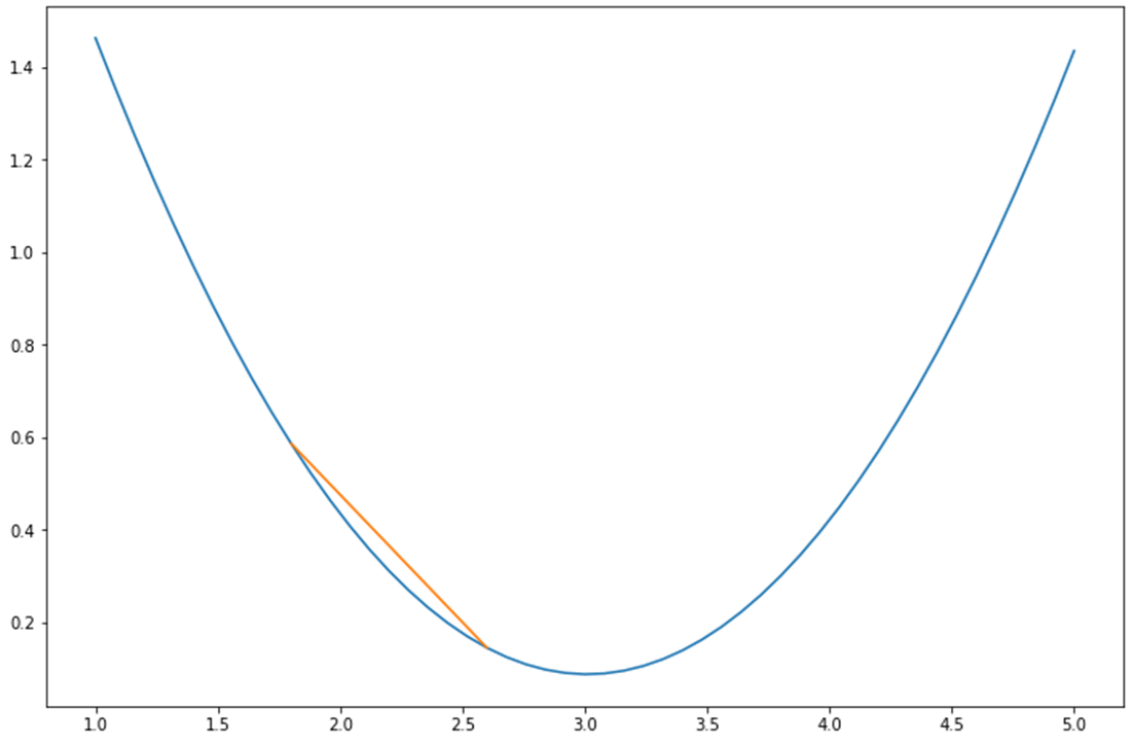


Aprendizaje por gradiente

Entrenando una regresión lineal

Observando el gráfico, intuitivamente se pude afirmar que:

- Si la pendiente es negativa, hay que incrementar el valor de w para disminuir el error.
- Si la pendiente es positiva, hay que decrementar el valor de w para disminuir el error.



Aprendizaje por gradiente

Entrenando una regresión lineal

Considerando lo anterior, podemos aplicar el siguiente algoritmo para aproximar el objetivo:

$$\underset{W,b}{\operatorname{argmin}}\, loss(Y, H(X\ w, b))$$

```
epoch = 0
w = random()
while epoch < MAX_EPOCH:
    epoch += 1
    #Calcular pendiente
    pw = pendiente_w(Y, X, w, b, delta)
    #Actualizar w
    w = w - lr * pw
```

Aprendizaje por gradiente

Entrenando una regresión lineal

Considerando lo anterior, podemos aplicar el siguiente algoritmo para aproximar el objetivo:

$$\underset{W,b}{\operatorname{argmin}}\, loss(Y, H(X\, w, b))$$

```
epoch = 0
w = random()
while epoch < MAX_EPOCH:
    epoch += 1
    #Calcular pendiente
    pw = pendiente_w(Y, X, w, b, delta)
    #Actualizar w
    w = w - lr * pw
```

Importante la inicialización random rompe simetrías

Hiper-parámetros:
- Epochs
- Delta
- Learning rate

Aprendizaje por gradiente

Entrenando una regresión lineal

Considerando lo anterior, podemos aplicar el siguiente algoritmo para aproximar el objetivo:

$$\underset{W,b}{\operatorname{argmin}}\, loss(Y, H(X\, w, b))$$

```
epoch = 0
w = random()
while epoch < MAX_EPOCH:
    epoch += 1
    #Calcular pendiente
    pw = pendiente_w(Y, X, w, b, delta)
    #Actualizar w
    w = w - lr * pw
```

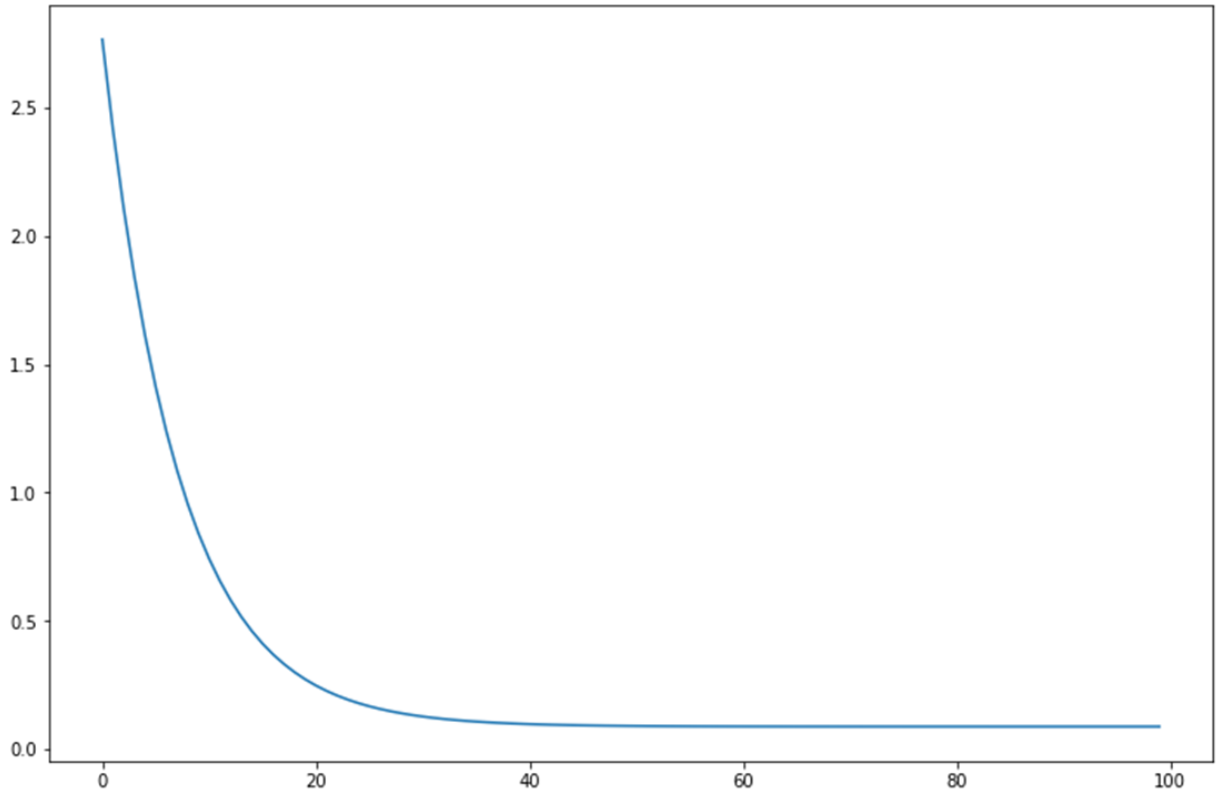
Hiper-parámetros:

- Epochs = 100
- Delta = 1e-6
- Learning rate = 0.1

Aprendizaje por gradiente

Entrenando una regresión lineal

- Aplicando este algoritmo, podemos verificar el valor del error en cada paso (epoch) del algoritmo
- Llegamos a que el valor final de w es 3.007618021813596



Aprendizaje por gradiente

Entrenando una regresión lineal

El algoritmo puede generalizarse para todos los parámetros de nuestro modelo. En este caso los parámetros son dos escalares w y b.

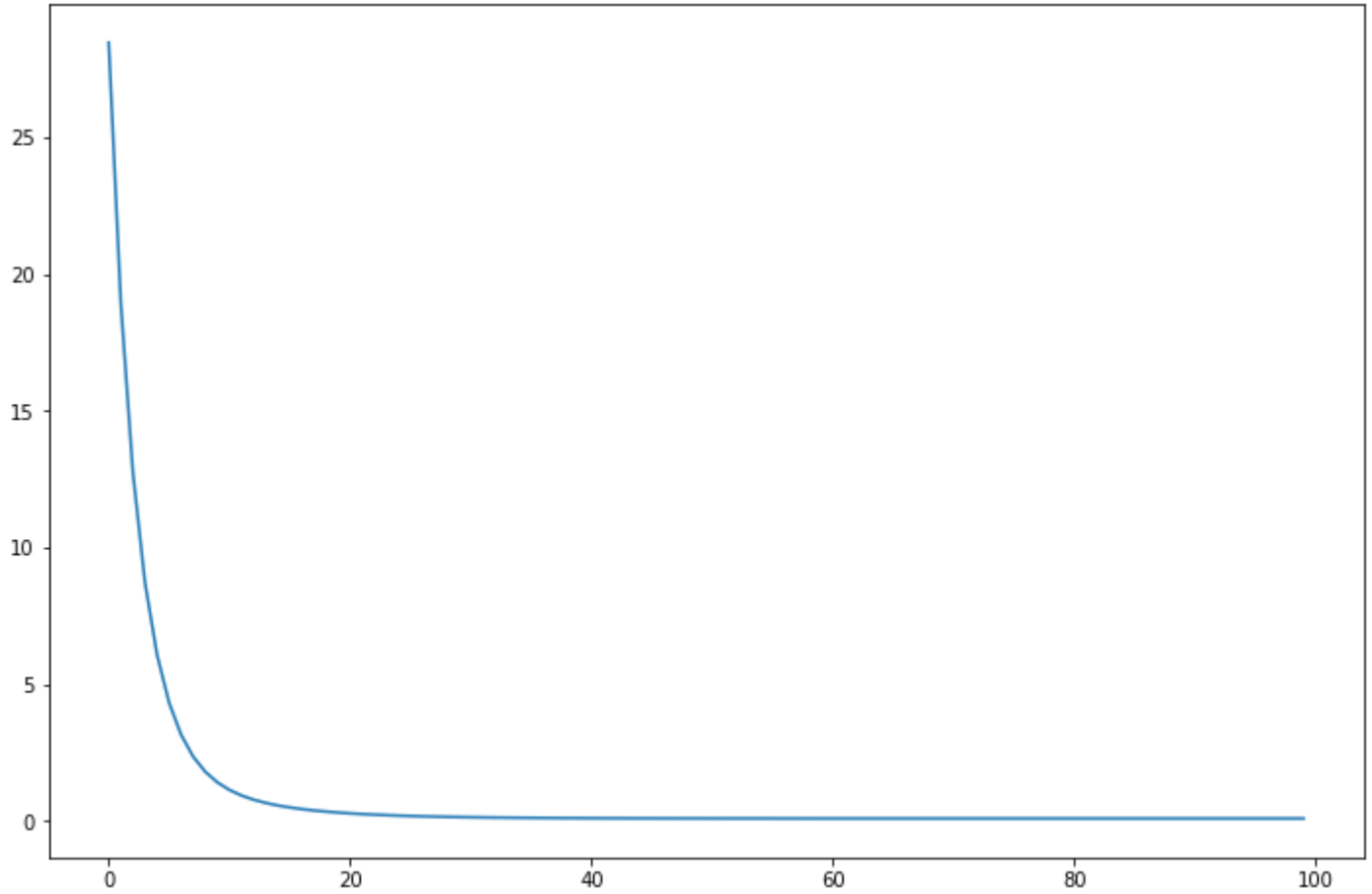
$$\underset{W,b}{\operatorname{argmin}}\, loss(Y, H(X\, w, b))$$

```
epoch = 0
w = random()
b = random()
while epoch < MAX_EPOCH:
    epoch += 1
    #Calcular pendiente
    pw = pendiente_w(Y, X, w, b, delta)
    pb = pendiente_b(Y, X, w, b, delta)
    #Actualizar w y b
    w = w - lr * pw
    b = b - lr * pb
```

Aprendizaje por gradiente

Entrenando una regresión lineal

En este caso, el valor final de w es 3.0076180218066573 y el valor de b es -0.02981975559479988



Aprendizaje por gradiente

Entrenando una regresión lineal

Considerando las pendientes, el valor deseado de delta es el menor posible. Por lo tanto:

$$\lim_{\Delta \to 0} pendiente_w(w, \Delta) = \lim_{\Delta \to 0} \frac{L(w) - L(w + \Delta)}{\Delta} = \frac{L(w)}{dw}$$

$$\lim_{\Delta \to 0} pendiente_b(b, \Delta) = \lim_{\Delta \to 0} \frac{L(b) - L(b + \Delta)}{\Delta} = \frac{L(b)}{db}$$

En nuestro caso, estas derivadas se puede resolver:

$$\frac{L(w)}{dw} = \frac{-2\sum\big(y - (xw + b)\big)x}{n}$$

$$\frac{L(b)}{db} = \frac{-2\sum(y - (xw + b))}{n}$$

Aprendizaje por gradiente

Entrenando una regresión lineal

El algoritmo puede generalizarse para todos los parámetros de nuestro modelo. En este caso los parámetros son dos escalares w y b.

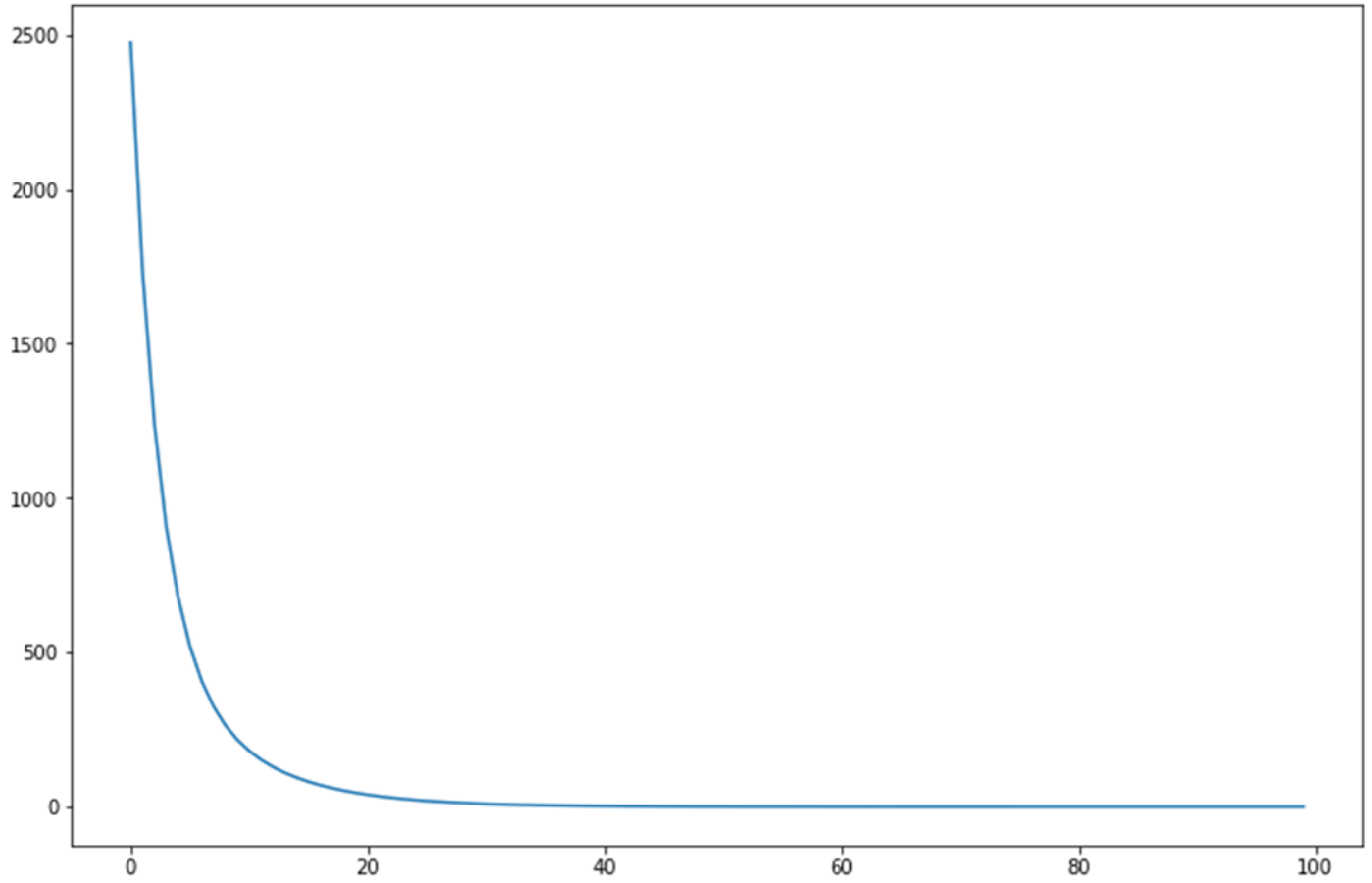
$$\underset{W,b}{\operatorname{argmin}}\, loss(Y, H(X\, w, b))$$

```
epoch = 0
w = random()
b = random()
while epoch < MAX_EPOCH:
    epoch += 1
    #Calcular gradientes
    gw, gb = gradientes(Y, X, w, b)
    #Actualizar w y b
    w = w - lr * gw
    b = b - lr * gb
```

Aprendizaje por gradiente

Entrenando una regresión lineal

En este caso, el valor final de w es 3.048557124852877 6 y el valor de b es -0.02981924793046757



TensorFlow

Librería de ML

TensorFlow

- TensorFlow es una librería de matemática orientada al ML.
- Desarrollada por Google y utilizada por diferentes empresas.
- Fácilmente utilizable desde Google Colab.
- Soporte para procesamiento en TPUs y GP-GPUs para acelerar la computación.
- Integra Keras, una librería para implementar redes neuronales de forma simplificada.

Implementando nuestro modelo y la función de perdida en TensorFlow.

```
import tensorflow as tf


def tf_lineal(x, w, b):
    return x * w + b

def tf_mse(y_true, y_pred):
    return tf.math.reduce_mean((y_true-y_pred)**2)

def tf_cost(y_true, x, w, b):
    return tf_mse(y_true, tf_lineal(x, w, b))
```

Implementando nuestro modelo y la función de perdida en TensorFlow.

```
#Tipos de datos esperados por tf
x = x.astype(np.float32)
y = y.astype(np.float32)
w = tf.random.uniform(shape=[], minval=-1, maxval=1)
b = tf.random.uniform(shape=[], minval=-1, maxval=1)
epochs = 100
lr = 0.1
errors = []
for i in range(epochs):
    with tf.GradientTape() as g:
        g.watch([w, b])
        loss = tf_cost(y, x, w, b)
        errors.append(loss.numpy())
    gw, gb = g.gradient(loss, [w, b])
    w = w - lr * gw
    b = b = - lr * gb
```

TensorFlow asume que el tipo de dato es np.float32

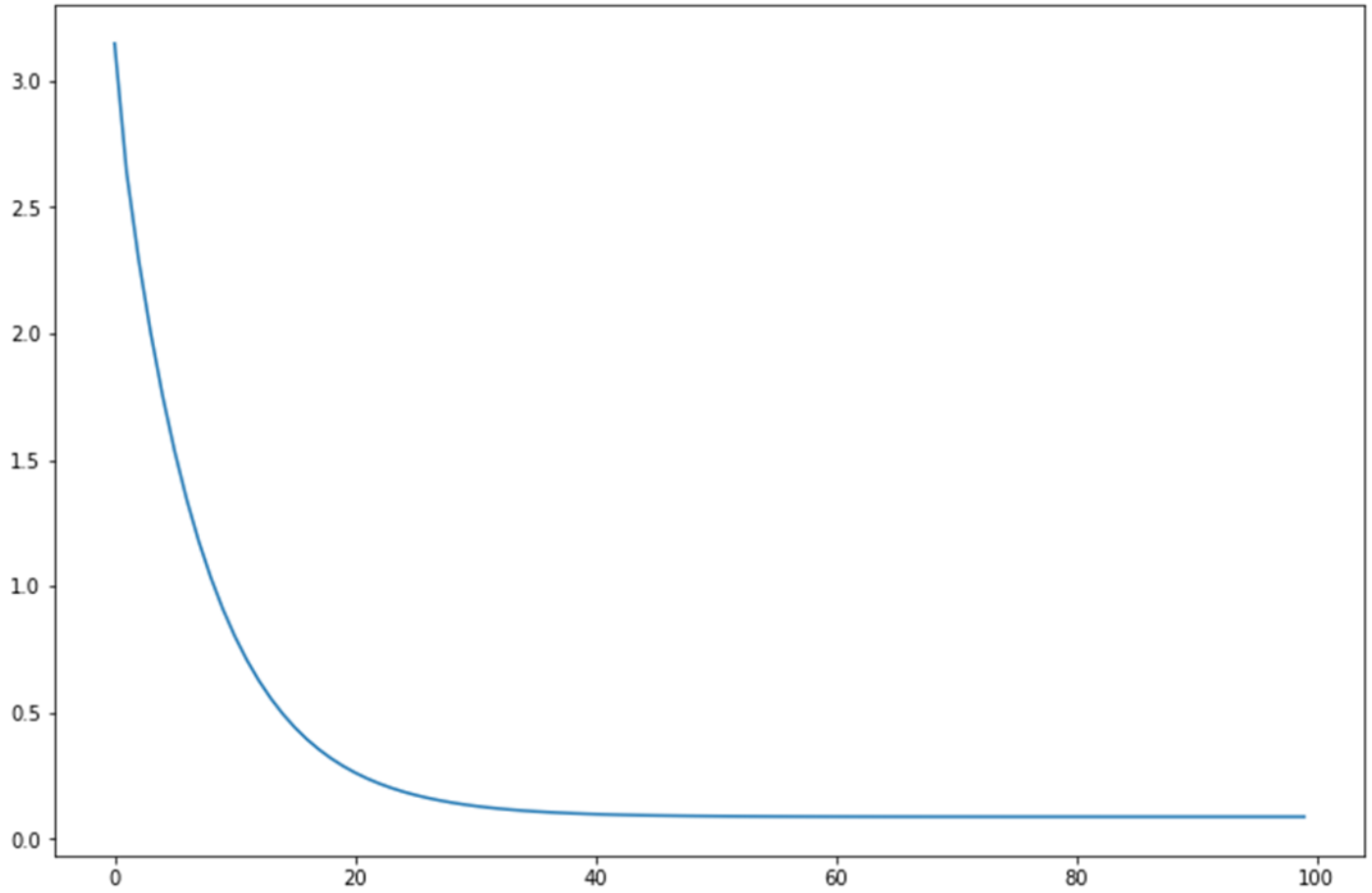
Inicializa los parámetros como arreglos tf

Graba los gradientes de las variables w y b

tensor.numpy() extrae el valor

Calcula los gradientes

La nueva implementación tiene un comportamiento similar al de la implementación en numpy.



Clasificando

Utilizaremos un dataset pequeño y real para ejemplificar algunas técnicas más avanzadas para entrenar utilizando el gradiente y explicar el porque de algunos preprocesamiento.
En particular, usaremos el Breast Cancer dataset de la Universidad de Wisconsin.
Dividiremos 500 para entrenamiento y 69 para testing.

Propiedad	Valor
Clases	2
Ejemplos por clase	212(M-0), 357(B-1)
Total de instancias	569
Dimensionalidad	30

Clasificando

El modelo lo podemos definir de la siguiente manera:

$$H(x) = \frac{1}{1 + e^{-(x \cdot W + b)}}$$

Y utilizamos el función de perdida conocida como *binary cross-entropy:*

$$H(Y, \hat{Y}) = \frac{\sum(-y.\log(\hat{y}) - (1 - y).\log(1 - \hat{y}))}{n}$$

Clasificando

El modelo lo podemos definir de la siguiente manera:

$$H(x) = \frac{1}{1 + e^{-(x \cdot W + b)}}$$

```
def logisticRegresion(x, w, b):
    return 1/(1+tf.exp(-(tf.matmul(x, w) + b)))[:,0]
```

Y utilizamos el función de perdida conocida como *binary cross-entropy:*

$$H(Y, \hat{Y}) = \frac{\sum(-y.\log(\hat{y}) - (1-y).\log(1-\hat{y}))}{n}$$

```
def crossentropy(yt, yp):
    return tf.math.reduce_mean(-yt*tf.math.log(yp)-(1-yt)*tf.math.log(1-yp))
```

Clasificando

Podríamos aplicar el mismo algoritmo que vimos para la regresión lineal:

```
w = tf.random.uniform(shape=[30, 1], minval=-1, maxval=1)
b = tf.random.uniform(shape=[], minval=-1, maxval=1)
epochs = 100
lr = 0.1
errors = []
for i in range(epochs):
    with tf.GradientTape() as g:
        g.watch([w, b])
        loss = crossentropy(y_train, logisticRegresion(x_train, w, b))
        errors.append(loss.numpy())
    gw, gb = g.gradient(loss, [w, b])
    w = w - lr * gw
    b = b = - lr * gb
```

Pero, probablemente todos los valores en W y b se vuelven NaN.

¿Por qué?

Pero, probablemente todos los valores en W y b se vuelven NaN.

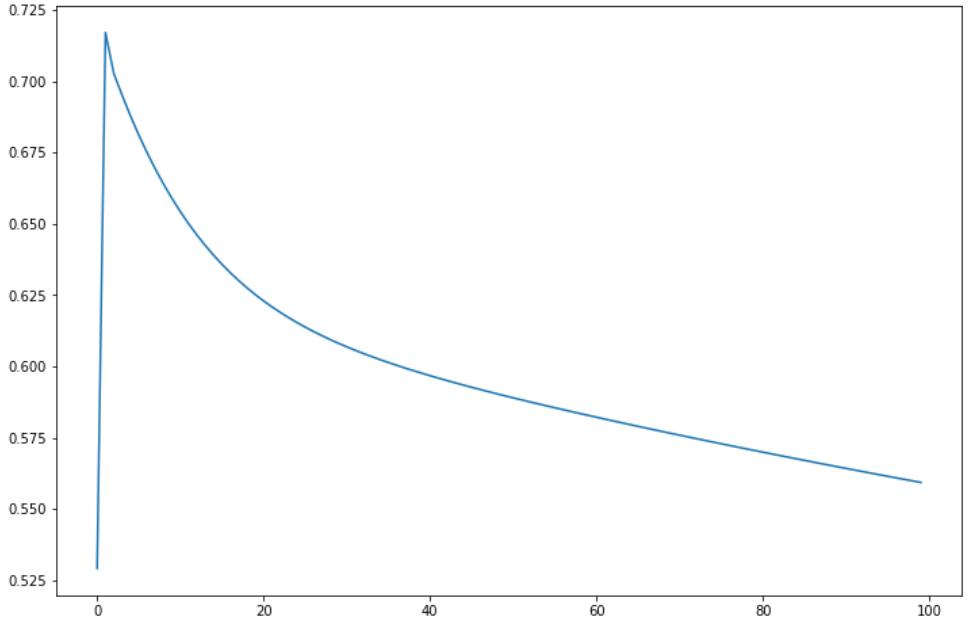
- Los atributos pueden tomar valores muy variados. Algunos llegan a valores de 50, mientras que otros no superan 1.
- Si bien la sigmoide es asintótica al 0 y 1 en el continuo, cuando se utiliza una representación de punto flotante, el valor se aproxima. Y $\log(0)$ es NaN.

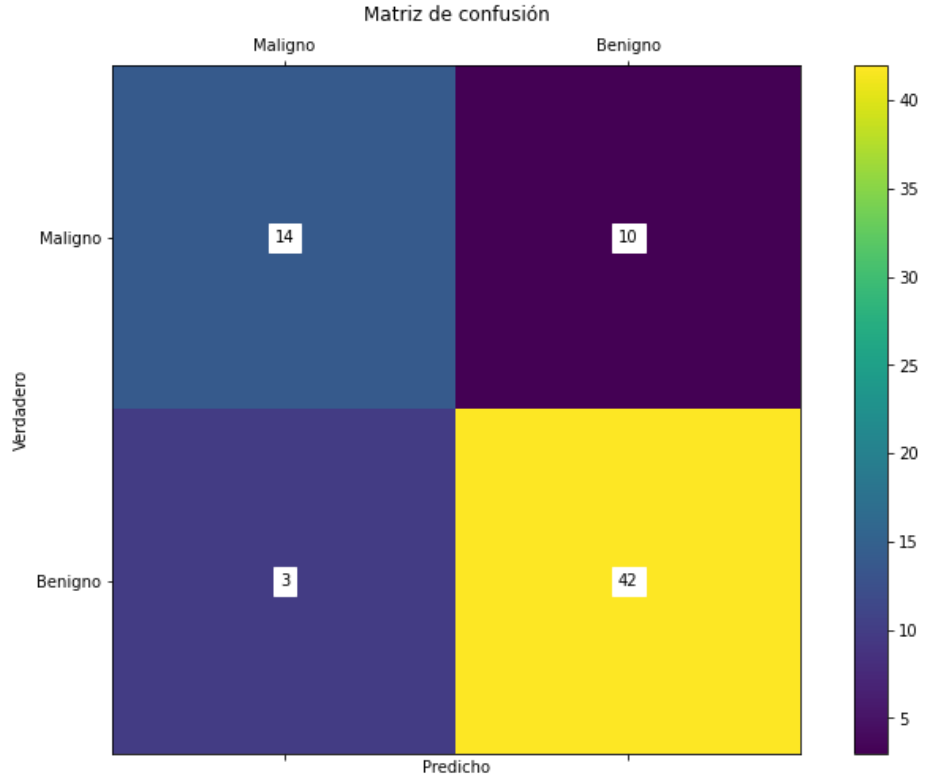
Solución:

- Acotar los valores de los atributos, por ejemplo escalándolos entre 0 y 1, o normalizándolos, o estandarizándolos.
- Esta solución facilita el aprendizaje, ya que utilizamos un solo *learning rate* para todas los parámetros.
- Finalmente, mientras más cercano a cero sea un número, más cercano será el próximo número, ya sea menor o mayor.

Clasificando

Escalando los atributos se pude entrenar con cierta efectividad. Sin embargo, para un buen entrenamiento necesita más de 100 epochs.





Stochastic Gradient Descent

Stochastic Gradient Descent es una aproximación de descenso por el gradiente para entrenar:

- Aproxima el gradiente de todo el dataset mediante actualizando sobre muestras aleatorias del mismo.
- Reduce la carga computacional ya que no require realizar operaciones sobre todo el dataset.
- En general, es más rápido, ya que actualiza los parámetros más frecuentemente.
- No en todos los pasos genera una mejor solución.

Stochastic Gradient Descent:

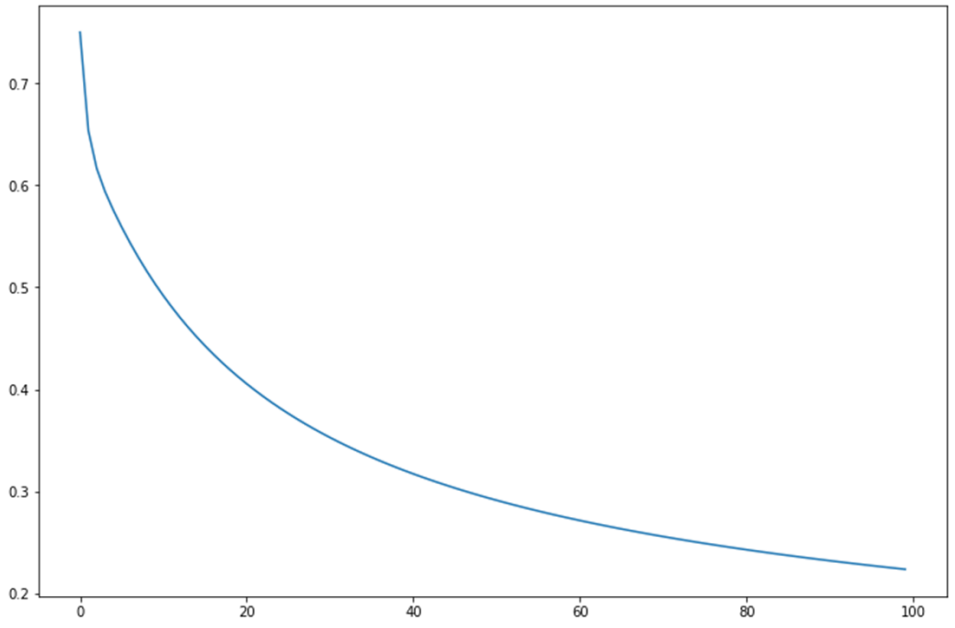
```
w = tf.random.uniform(shape=[30, 1], minval=-1, maxval=1)
b = tf.random.uniform(shape=[], minval=-1, maxval=1)
epochs = 100
lr = 0.1
for i in range(epochs):
    x_s, y_s = shuffle(x_train, y_train)
    for mb in range(0, 500, 50):
        with tf.GradientTape() as g:
            g.watch([w, b])
            loss = loss_f(y_s[mb:mb+50], x_s[mb:mb+50], w, b)
        gw, gb = g.gradient(loss, [w, b])
        w = w - lr * gw
        b = b - lr * gb
```

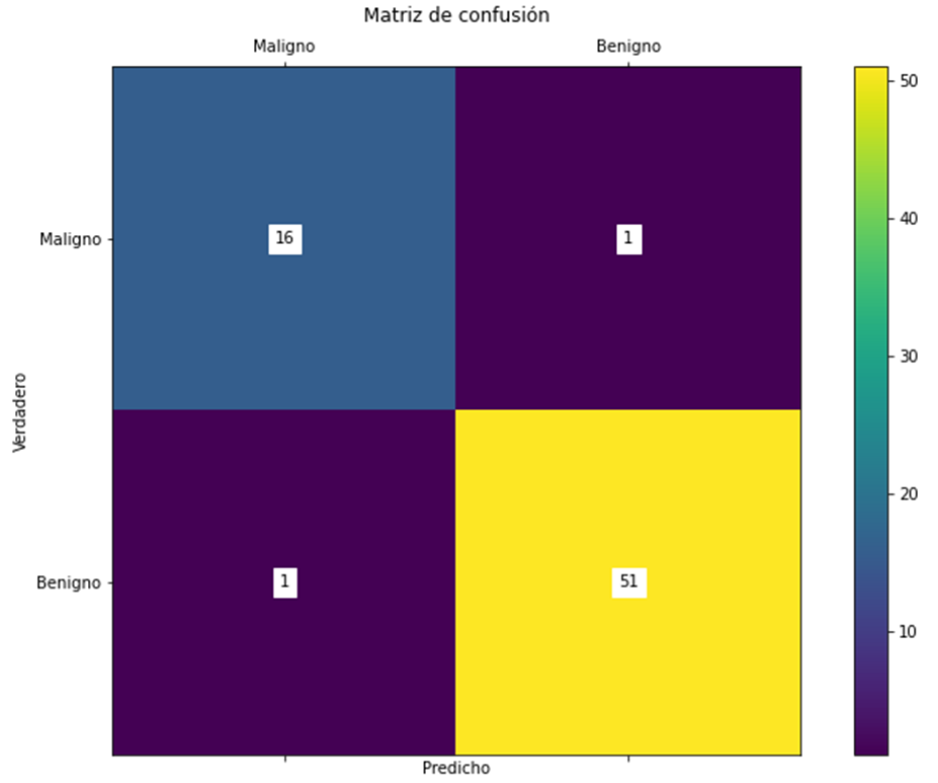
Desordenar las instancias para el entrenamiento

Actualizar los pesos por cada mini batch

Stochastic Gradient Descent

Con esta técnica se mejoran notablemente los resultados de entrenamiento…

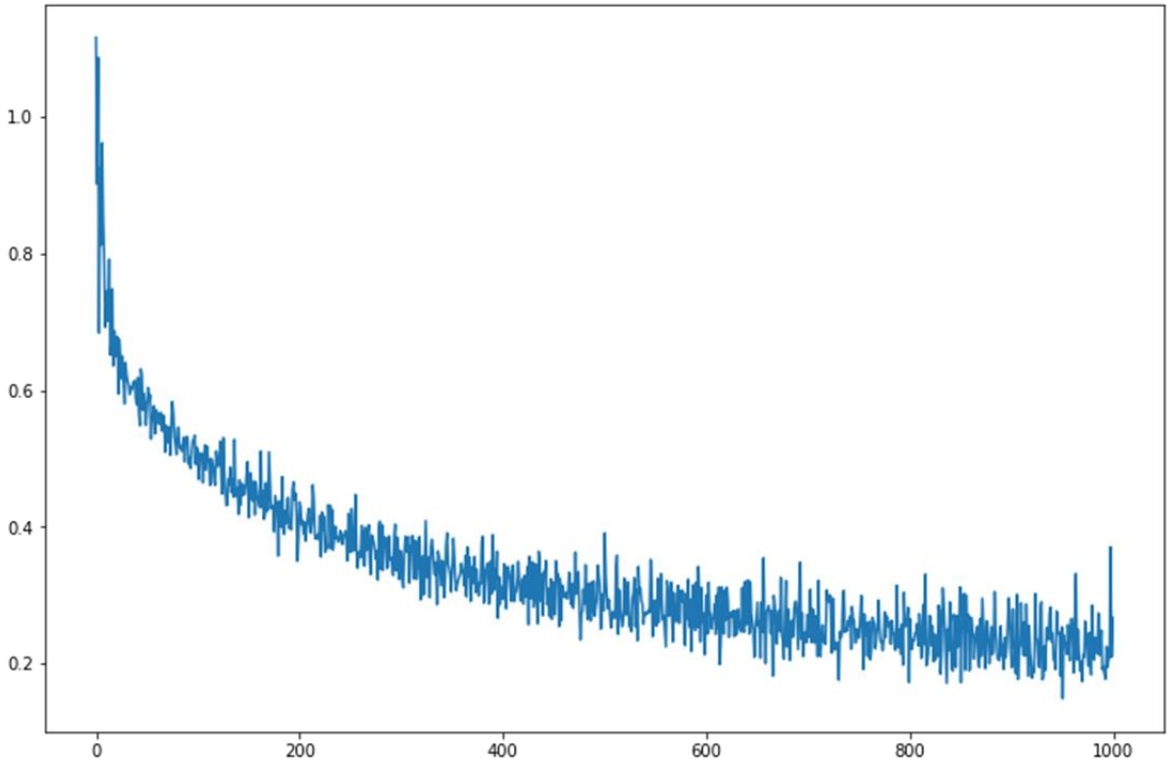




Stochastic Gradient Descent

Mirando en más detalle el entrenamiento vemos que:

- La perdida para cada mini batch varía notablemente cuando se lo compara con sus vecinos.
- Si bien no es monótono, hay una clara tendencia a que la perdida disminuya en el tiempo.



Stochastic Gradient Descent: Momentum

Momentum es una técnica para mejorar el entrenamiento basado en que:

- No todos los mini batchs aportan al entrenamiento.
- En general, hay una tendencia a disminuir el error.
- Si se considera el histórico, se tiene mejor información de cuánto y en qué dirección se deben actualizar los parámetros.
- Actualizaciones más recientes deben tener más importancia ya que los parámetros se actualizan con cada mini batch.
- Actualizaciones que ocurrieron hace mucho no deberían tener un peso significante en nuevas actualizaciones.
- Las actualizaciones se hacen indirectamente a través mediante velocidades, las cuales son actualizadas con cada mini batch.

Stochastic Gradient Descent: Momentum

Momentum:

```
vw = tf.zeros(shape=[30, 1])
vb = tf.zeros(shape=[])
momentum = 0.9
for i in range(epochs):
    x_s, y_s = shuffle(x_train, y_train)
    for mb in range(0, 500, 50):
        with tf.GradientTape() as g:
            g.watch([w, b])
            loss = loss_f(y_s[mb:mb+50], x_s[mb:mb+50], w, b)
        gw, gb = g.gradient(loss, [w, b])
        vw = momentum * vw - lr * gw
        vb = momentum * vb - lr * gb
        w = w + vw
        b = b + vb
```

Velocidades iniciales son cero

Momentum: peso del factor histórico

Actualización de la velocidad basado en los gradientes

Actualización de los pesos basados en la velocidad actualizada

Stochastic Gradient Descent: Momentum

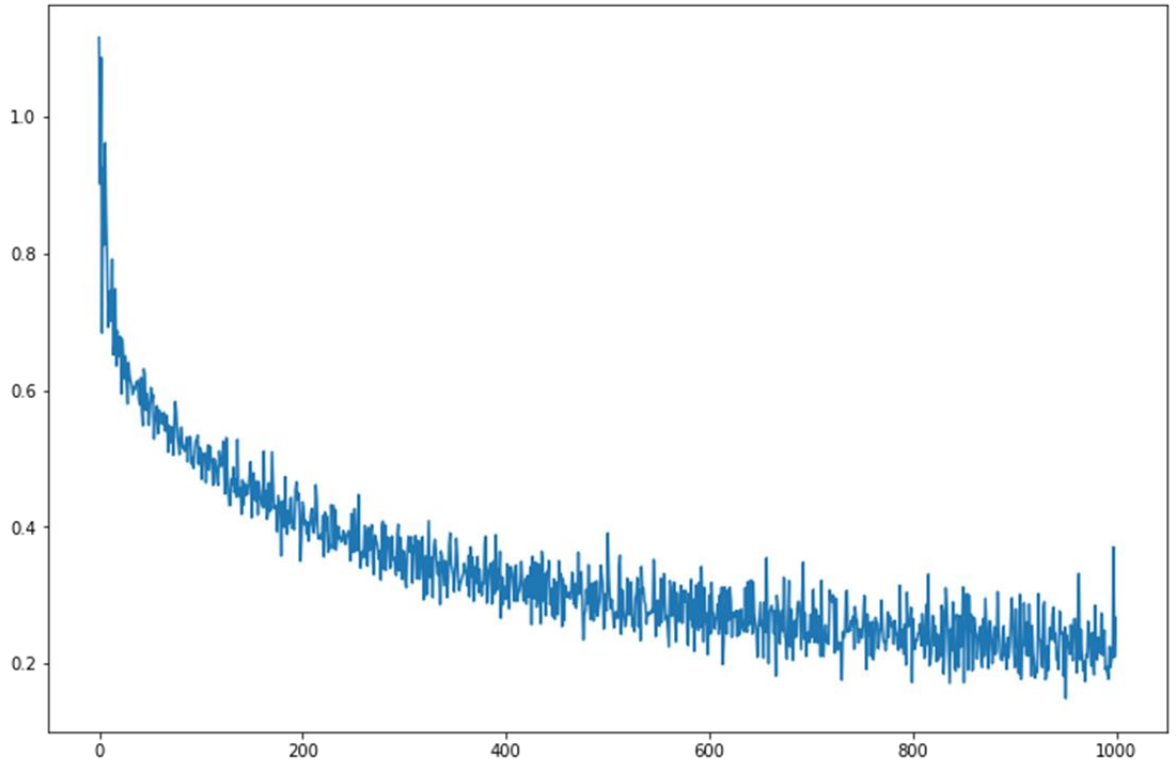
Con esta técnica vuelven a aparecer los NaN, ya que como se mejoran los valores, las predicciones se acercan a 0 o 1. Para evitar este problema se suele recortar los valores extremos.

```
def crossentropy(yt, yp):
    return tf.math.reduce_mean(-yt*tf.math.log(tf.clip_by_value(yp, 1e-6, 1)) -
                               (1-yt)*tf.math.log(tf.clip_by_value(1-yp, 1e-6, 1)))
```

Stochastic Gradient Descent: Momentum

Entrenamiento sin momentum:

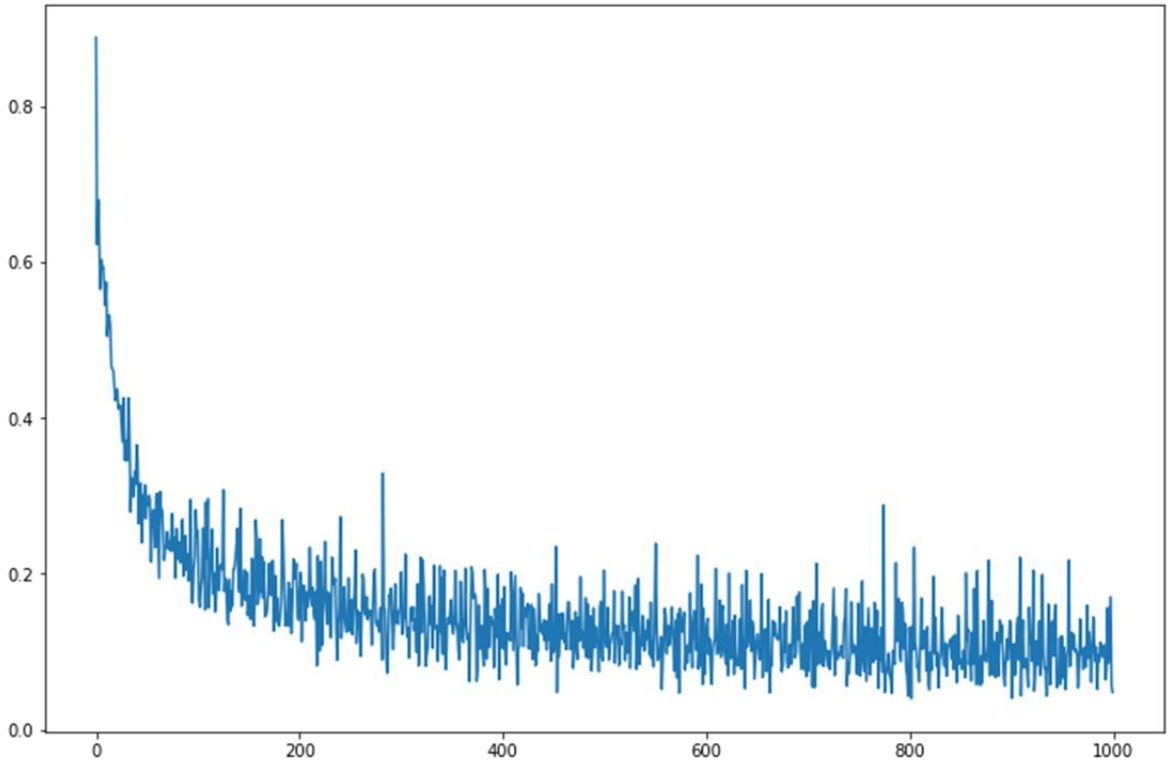
¿Se podrá ver la mejora?



Stochastic Gradient Descent: Momentum

Ventajas del momentum:

- El entrenamiento es más rápido.
- Se llegan a menores valores de perdida.
- Mejora la convergencia.
- No agrega complejidad computacional.

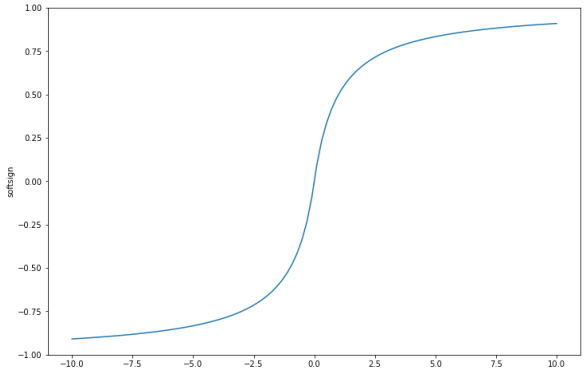


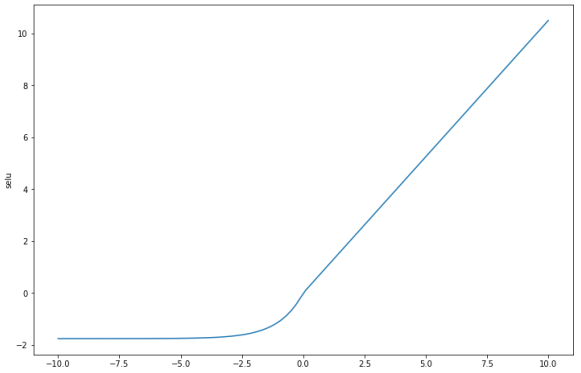
Funciones de Activación

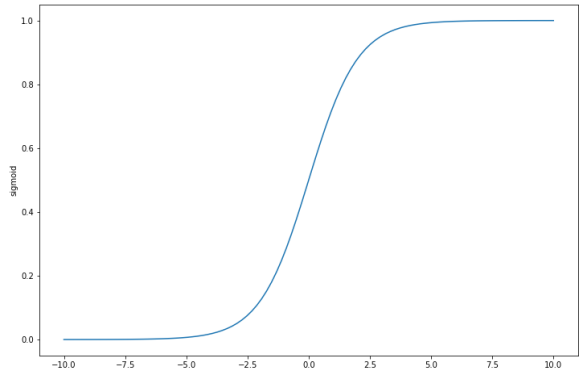
Softmax, tanh y relu

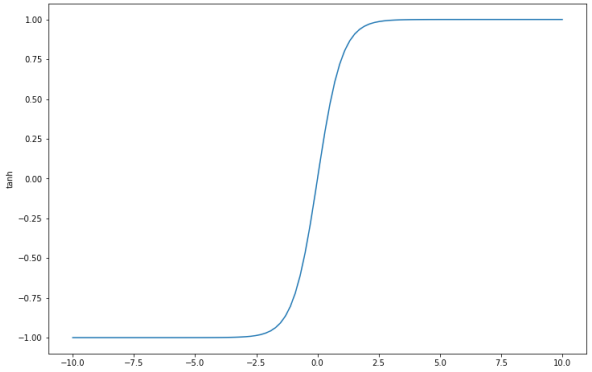
Otras funciones de activación que encontraremos, además de la sigmoide e identidad, son:

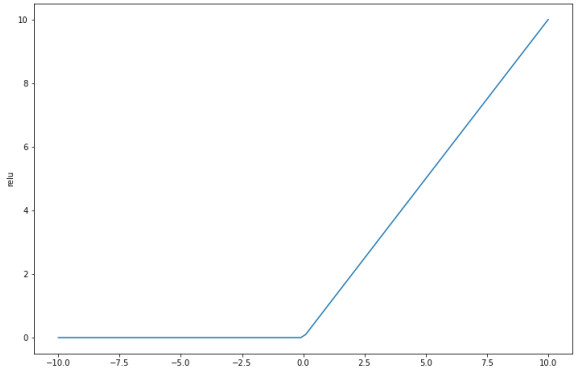
- Tanh
- Relu
- Selu
- Softplus
- Softsign
- Softmax

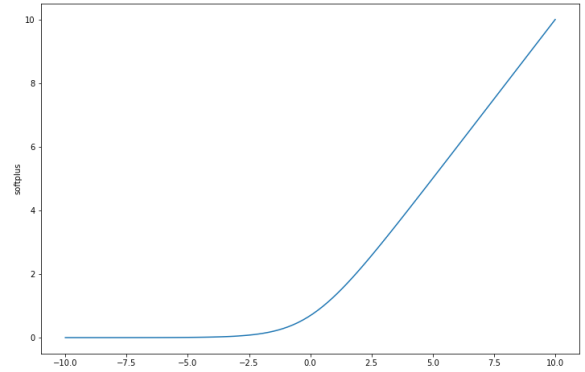












Funciones de Activación

Softmax

Softmax es una función especial, ya que no se aplica elemento, sino que se aplica de forma vectorial.

- Usada para clasificación multi-clase.
- Asigna un valor entre 0 y 1 a cada elemento del vector.
- La suma total de los elementos es 1.
- Usada para asignar probabilidad a las distintas clases.

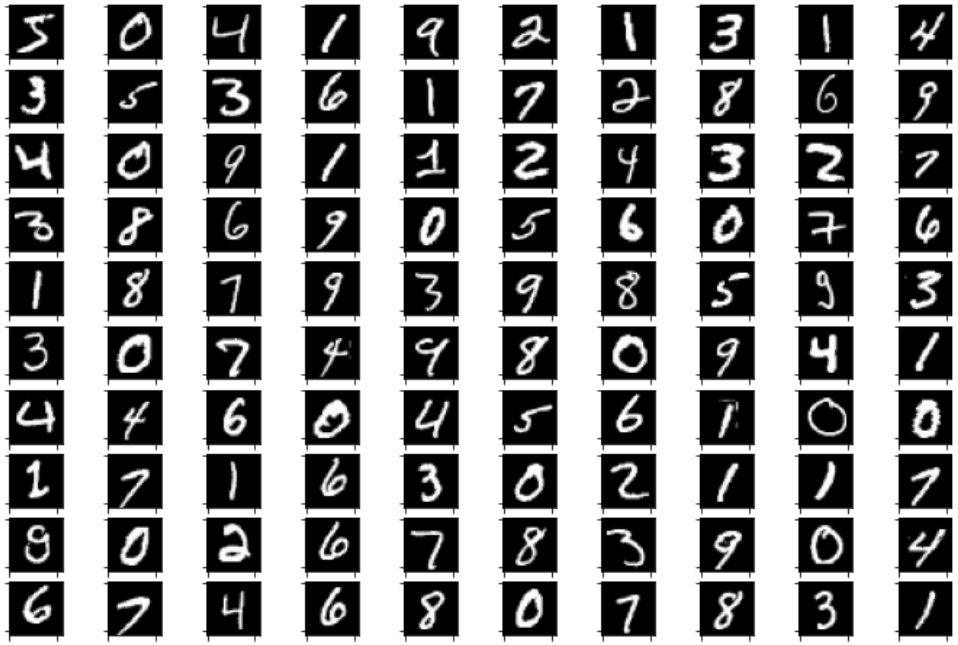
$$softmax(Z)_i = \frac{e^{z_i}}{\sum e^{z_j}}$$

Clasificación multiclase

Softmax

Ejemplo el dataset MNIST

Propiedad	Valor
Clases	10
Tamaño de las imagenes	28 X 28 (784)
Instancias de entrenemiento	60.000
Instancias de entrenemiento	10.000
Valor mínimo de cada pixel	0
Valor máximo de cada pixel	255



Clasificación multiclase

Softmax

Ejemplo el dataset MNIST

- Las instancias están representadas por matrices de 28x28.
  - Podemos mapearlo a vectores aplanando las matrices poniendo las filas una atrás de otra.
- Cada pixel tiene un valor entre 0 y 255.
  - Podemos dividirlos por 255 para llevar los valores al rango [0-1]
- Las clases están representadas por un número entre 0 y 9.
  - Para adaptar a la salida de una softmax debemos cambiar la representación a one-hot encoding.
  - Ejem:
    - $2 \rightarrow (0,0,1,0,0,0,0,0,0,0)$
    - $8 \rightarrow (0,0,0,0,0,0,0,0,1,0)$

Ejemplo el dataset MNIST

```
size = x_train.shape[1]*x_train.shape[2]
x_train = x_train.reshape((x_train.shape[0], size)) / 255
x_test = x_test.reshape((x_test.shape[0], size)) / 255

yc_train, yc_test = to_categorical(y_train), to_categorical(y_test)

x_train = x_train.astype(np.float32)
x_test = x_test.astype(np.float32)
yc_train = yc_train.astype(np.float32)
yc_test = yc_test.astype(np.float32)
```

Ejemplo el dataset MNIST

Adaptación de la función de pedida:

- Para clasificación de perdida crossentropy pude usarse, pero la versión vista antes es binaria, o sea se calcularía elemento a elemento.
- En el modelo, cada elemento del vector de probabilidades depende de los otros elemento.

$$L(Y,\hat{Y}) = \frac{-\sum y \log \hat{y}}{n} = \frac{-(0,..,1,..0)\log(\widehat{y_1}) - \cdots - (0,..,1,..0)\log(\widehat{y_n})}{n}$$

Ejemplo el dataset MNIST

```
def softmax(z):
    exp = tf.exp(z)
    return exp / tf.expand_dims(tf.math.reduce_sum(exp, axis=1), axis=-1)

def predict(x, w, b):
    return softmax(tf.matmul(x, w) + b)

def categorical_crossentropy(y_true, y_pred):
    return tf.math.reduce_mean(-tf.math.reduce_sum(y_true *
                tf.math.log(tf.clip_by_value(y_pred, 1e-6, 1)), axis=1))
```

Clasificación multiclase

Softmax

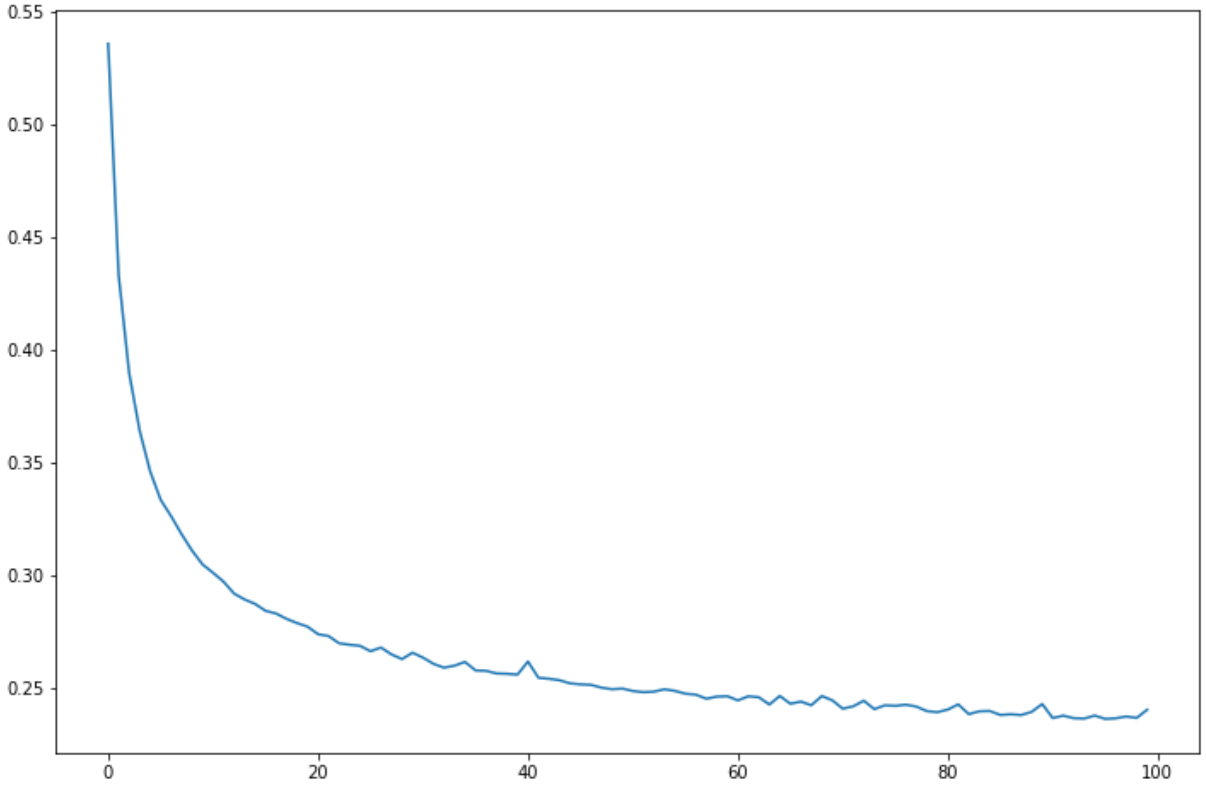
Ejemplo el dataset MNIST

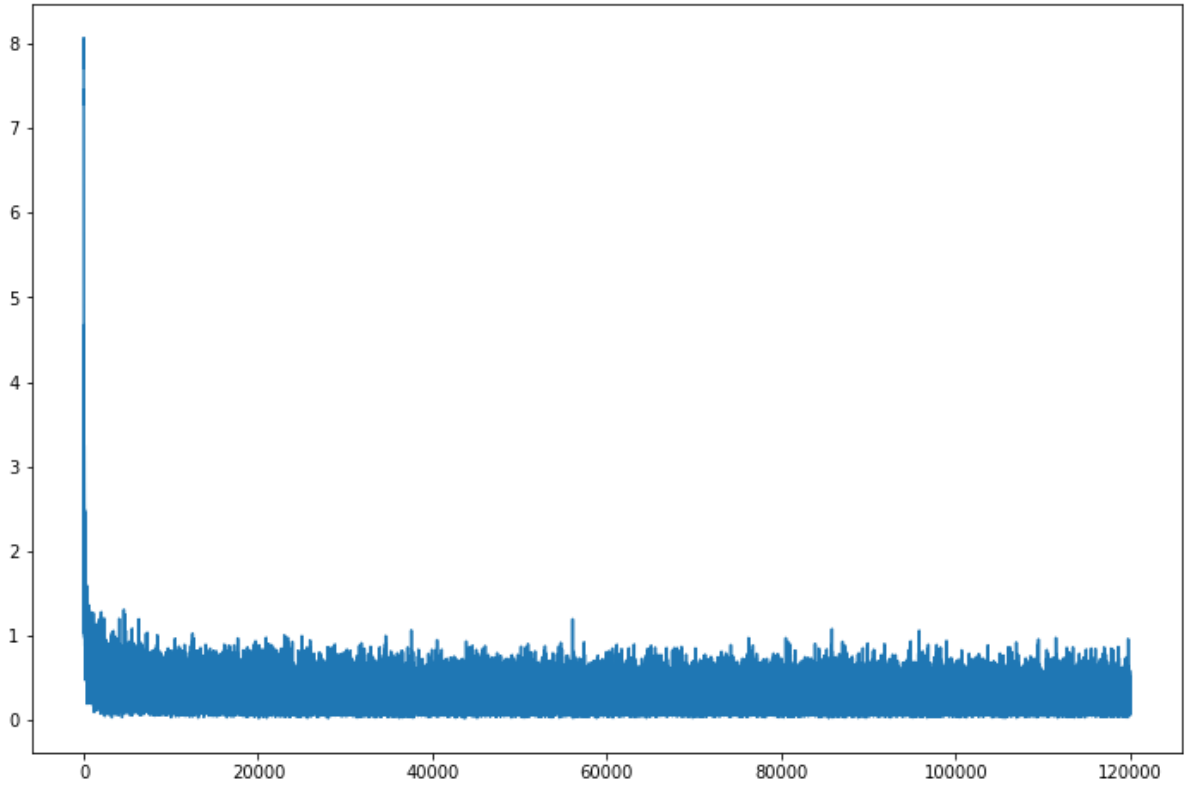
```
w = tf.random.uniform(shape=[size, 10], minval=-1, maxval=1)
b = tf.random.uniform(shape=[10], minval=-1, maxval=1)
vw = tf.zeros(shape=[size, 10])
vb = tf.zeros(shape=[10])
for i in tqdm(range(epochs)):
    x_s, y_s = shuffle(x_train, yc_train)
    for mb in range(0, x_s.shape[0], 50):
        with tf.GradientTape() as g:
            g.watch([w, b])
            loss = loss_f(y_s[mb:mb+50], x_s[mb:mb+50], w, b)
        gw, gb = g.gradient(loss, [w, b])
        vw = momentum * vw - lr * gw
        vb = momentum * vb - lr * gb
        w = w + vw
        b = b + vb
```

Clasificación multiclase

Softmax

Ejemplo el dataset MNIST

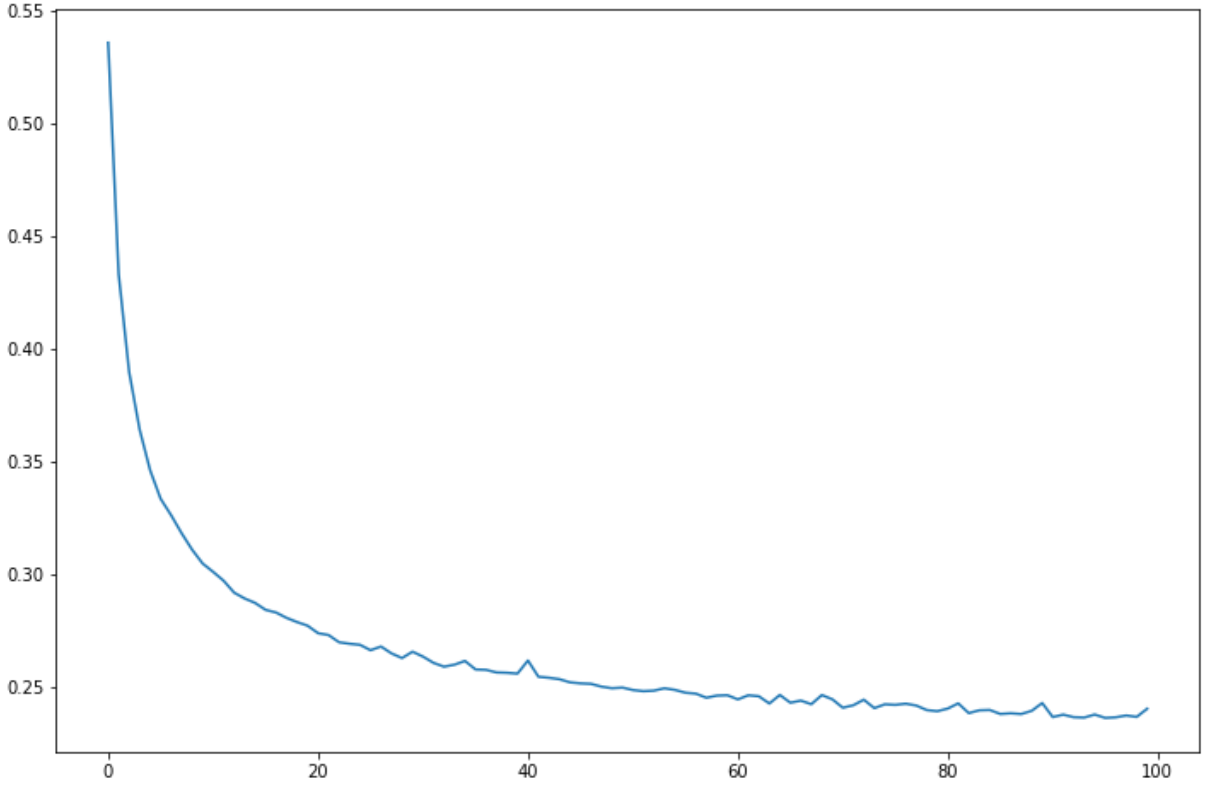


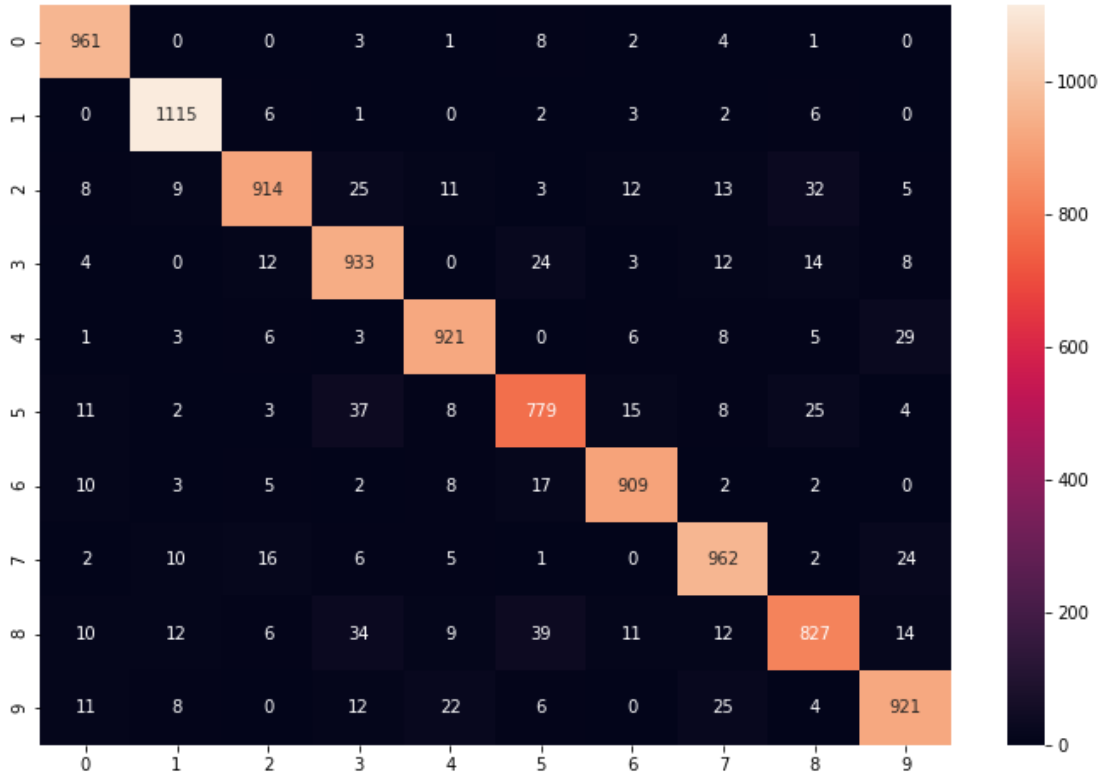


Clasificación multiclase

Softmax

Ejemplo el dataset MNIST





Clasificación multiclase

Uso de variables (optimización)

Ejemplo el dataset MNIST

```
w = tf.Variable(tf.random.uniform(shape=[size, 10], minval=-1, maxval=1))
b = tf.Variable(tf.random.uniform(shape=[10], minval=-1, maxval=1))
vw = tf.Variable(tf.zeros(shape=[size, 10]))
vb = tf.Variable(tf.zeros(shape=[10]))
for i in tqdm(range(epochs)):
    x_s, y_s = shuffle(x_train, yc_train)
    for mb in range(0, x_s.shape[0], 50):
        with tf.GradientTape() as g:
            g.watch([w, b])
            loss = loss_f(y_s[mb:mb+50], x_s[mb:mb+50], w, b)
        gw, gb = g.gradient(loss, [w, b])
        vw.assign(momentum * vw - lr * gw)
        vb.assign(momentum * vb - lr * gb)
        w.assign(w + vw)
        b.assign(b + vb
```

Conclusiones

Resumen

- Concepto de perceptrón.
- Repaso de los conceptos de Regresión Lineal y Regresión Logística.
- Repaso funciones de perdida clásicas.
- Aprendizaje por el gradiente.
  - Stochastic Gradiant Descent
  - Momentum
- Mencionamos algunas funciones de activación (que encontraremos más adelante).
- Utilizamos SDG+Momentum para entrenar un clasificador multiclase:
  - Función Softmax.
  - Función de perdida Categorical CrossEntropy.