2. FUNDAMENTOS DE APRENDIZAJE AUTOMÁTICO

Como contexto del concepto de redes neuronales, se partirá de la noción de regresión lineal. Una regresión implica modelar la relación entre variables independientes y una variable dependiente con el fin de predecir un valor numérico. Cuando se habla de regresión lineal, esta relación entre variables independientes y la variable dependiente es lineal, es decir se puede representar como una suma ponderada de las variables independientes hacia la variable dependiente.

Consideremos un conjunto de datos de entrenamiento, que incluye *n* ejemplos, donde el i-ésimo ejemplo se representará por el superíndice *i*. Las variables independientes o atributos de los datos se representarán por **x**, donde x_1, x_2, … corresponden al primer y segundo atributo, respectivamente. De esta forma, el i-ésimo ejemplo se representa de la forma:

$$x^i = \begin{bmatrix} x_1^i \\ x_2^i \end{bmatrix} \tag{1}$$

Y la variable independiente, o etiqueta del i-ésimo ejemplo se representa de la forma:

$$y^i \tag{2}$$

Utilizando esta representación, un conjunto de datos queda estructurado de la siguiente forma:

	Atributo 1	**Atributo 2**	⋯	**Atributo n**	**Etiqueta Salida (y)**
x^1	x_1^1	x_2^1	⋯	x_n^1	y^1
x^2	x_1^2	x_2^2	⋯	x_3^2	y^2
x^3	x_1^3	x_2^3	⋯	x_n^3	y^3
x^4	x_1^4	x_2^4	⋯	x_n^4	y^4

x^5	x_1^5	x_2^5	$\cdots$	x_n^5	y^5
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
x^m	x_1^m	x_2^m	$\cdots$	x_n^m	y^m

Siendo m el número de ejemplos o muestras del dataset, y n el número de atributos o variables independientes.

Para contextualizar lo anterior, tomemos un ejemplo relacionado con un modelo de regresión para estimar el valor de una vivienda en una ciudad determinada. Consideremos cuatro atributos de los datos entrada: el área, la antigüedad, el número de habitaciones y la zona donde está ubicada. Bajo este contexto, un modelo de predicción lineal del precio de la vivienda quedaría expresado bajo una suma ponderada, así:

$$precio = w_a \times área + w_e \cdot edad + w_h \cdot hab + w_z zona + b \quad (3)$$

En esta suma ponderada, los términos w_x corresponden al peso que tiene un determinado atributo en la suma, es decir qué tanto aporta en el cálculo de la variable independiente, que en este caso es el precio del bien inmueble. Por ejemplo, w_a representa el aporte o peso que tiene la superficie de la vivienda en el cálculo del precio.

Además, esta suma ponderada incluye un valor de sesgo o *bias* que permite ajustar el modelo. Para entender este concepto, podemos relacionarlo con el nivel DC que puede tener la respuesta de un circuito eléctrico, o de forma general podemos imaginar un modelo de regresión lineal con una sola variable independiente representado a través de la ecuación de una recta, expresada en su forma pendiente-intercepto:

$$Y = mx + b \quad (4)$$

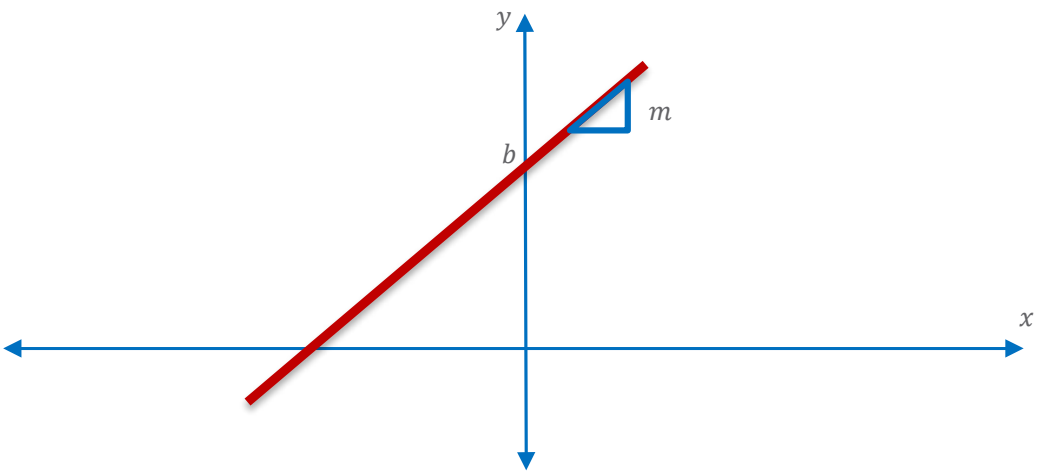

Figura 3. Función lineal representada en forma pendiente-intercepto

Si relacionamos la ecuación de la recta con el modelo de regresión lineal, la pendiente (m) corresponde al peso de la variable independiente (x) y el intercepto corresponde al sesgo o *bias* necesario para ajustar la recta en el eje vertical (y). De esta forma podríamos reescribir un modelo de regresión lineal de una sola variable independiente como:

$$y = wx + b \tag{5}$$

Donde w lo asociamos a la pendiente y b al intercepto de la recta.

Al expresar el modelo de regresión lineal como una suma ponderada, el objetivo será encontrar $\boldsymbol{w}$ y b a partir de los ejemplos del conjunto de datos, de tal forma que dichos valores se ajusten lo mejor posible a la etiqueta o valor de salida de los datos.

GENERALIZACIÓN

Considerando un caso que involucre n atributos (x_1, x_2, … x_n) en los datos de entrada, el modelo de regresión lineal para una muestra del conjunto de datos se puede expresar como:

$$\hat{y} = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b \tag{6}$$

Donde $\hat{y}$ es el valor de salida estimado por el modelo.

De esta forma, la dimensión del vector de atributos ***x*** de un ejemplo es *n*-dimensional al igual que la dimensión del vector de pesos, ***w***, es decir:

$$\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{w} \in \mathbb{R}^n \tag{7}$$

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} + b \tag{8}$$

Al considerar el grupo de atributos de entrada como un vector ***x*** (x_1, x_2, … x_n) y el grupo de pesos como un vector **w** (w_1, w_2, … w_n), el modelo de regresión se puede expresar como:

Ahora, considerando que los datos de entrada están consolidados en un dataset que incluye *m* ejemplos, estructurados en una fila por cada ejemplo y una columna por cada atributo, es decir el conjunto de *n* atributos para *m* ejemplos será una matriz **X** de dimensiones:

$$\boldsymbol{X} \in \mathbb{R}^{m \times n} \tag{9}$$

Dado que la salida normalmente es un valor único, esta será un vector que involucre los *m* valores de salida (uno para cada ejemplo), es decir:

$$\hat{y} \in \mathbb{R}^m \tag{10}$$

Por su parte, las dimensiones del vector de pesos se mantienen con respecto al vector con un solo ejemplo, ya que el proceso de regresión lineal obtendrá un solo modelo o comportamiento para todos los ejemplos, es decir:

$$\boldsymbol{w} \in \mathbb{R}^n \tag{11}$$

De esta forma, el modelo de regresión lineal se puede generalizar para *m* ejemplos con *n* atributos así:

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w} + b \tag{12}$$

Donde $\hat{y}$ corresponde al valor estimado por el modelo (salida), **X** corresponde a una matriz que contiene los *n* atributos de entrada para *m* ejemplos y *b* corresponde al sesgo o ajuste del modelo.

El objetivo del entrenamiento del modelo generalizado sigue siendo el mismo: encontrar valores para ***w*** y b, que permitan realizar predicciones sobre nuevos datos (muestras a partir de la misma distribución de **X**) con el menor error posible.

FUNCIÓN DE PÉRDIDA

Dado que el objetivo del entrenamiento del modelo a partir de los datos involucra obtener los parámetros (***w*** y b) que presenten el menor error en la predicción, es necesario utilizar una función que permita medir dicho error. Esta función se conoce como función de pérdida, y se encarga de cuantificar la distancia entre el valor real (y) y el estimado por la función objetivo ($\hat{y}$).

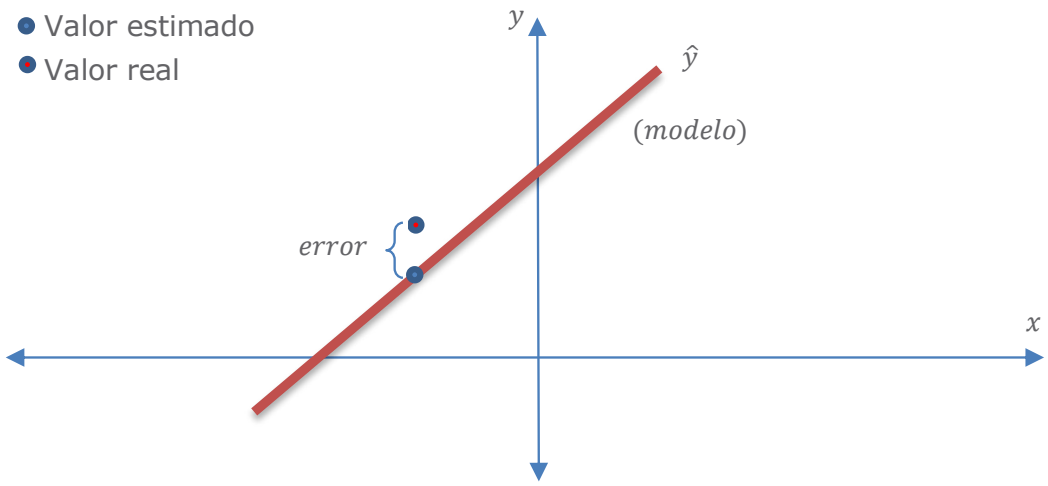

Figura 4. Error entre valor estimado y valor real

Típicamente, para calcular una función de error en aprendizaje automático es posible utilizar una métrica de distancia entre dos vectores, o alguna alternativa para calcular el tamaño de un vector. Por ejemplo, la función de error cuadrático o norma L_2:

$$l^i(\boldsymbol{w}, b) = \frac{1}{2}\left(\hat{y}^i - y^i\right)^2 \tag{13}$$

De esta forma, para cada uno de los ejemplos (i), la función calcula el cuadrado de la diferencia entre el valor estimado respecto al valor real. Dado el exponente de la expresión, esta función penaliza en mayor medida valores altos en la

diferencia, dada su naturaleza cuadrática. La expresión anterior también utiliza la constante ½, que permite alcanzar un valor unitario en el coeficiente cuando se derive la función al calcular el gradiente (lo que se verá reflejado con mayor claridad en las siguientes ecuaciones).

Dado que la ecuación anterior determina el valor de la función de pérdida para un ejemplo dado, y en aras de generalizar la función de pérdida para todos los ejemplos del conjunto de datos, se realiza un promedio sobre el valor de la pérdida para todos y cada uno de los ejemplos, así:

$$L(\boldsymbol{w}, b) = \frac{1}{m}\sum_{i=1}^{m} l^i(\boldsymbol{w}, b) = \frac{1}{m}\sum_{1}^{m}\left(\boldsymbol{w}^T\boldsymbol{x}^i + b - y^i\right)^2 \quad (14)$$

De esta forma, el objetivo del entrenamiento del modelo es minimizar la función de pérdida respecto a **w** y b. Es decir, encontrar los parámetros $\boldsymbol{w}$ y b en el dominio de la función L en los que el valor de esta se minimiza.

$$w^*, b^* = \underset{w,b}{argmin} L(\boldsymbol{w}, b) \quad (15)$$

REDUCCIÓN DEL ERROR

La reducción de error en la función de pérdida con respecto a los pesos ($\boldsymbol{w}$) y el sesgo o bias (b), involucra la actualización de los valores de w y b hacia la dirección del mínimo de la función.

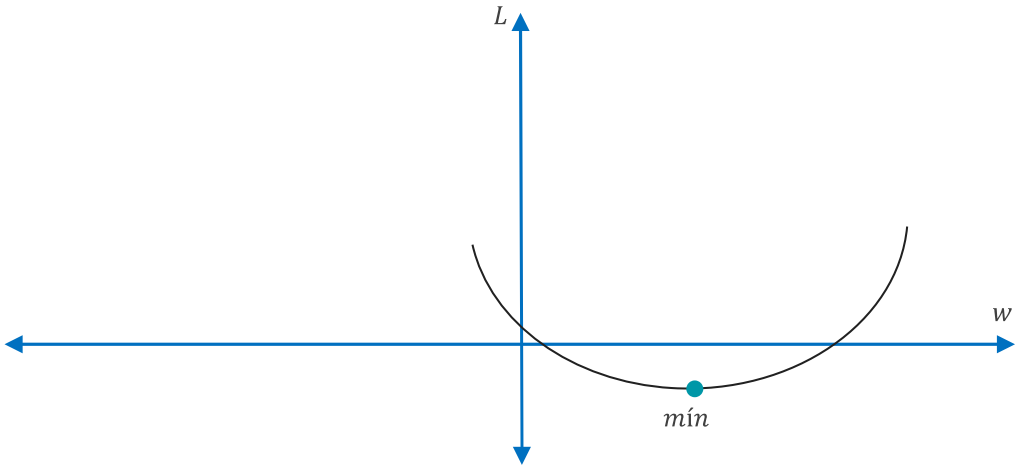

Figura 5. Ilustración del mínimo de una función

Este proceso se alcanza mediante un algoritmo de optimización como el algoritmo de gradiente descendente, el cual reduce iterativamente el error mediante la actualización de los parámetros en la dirección que disminuye progresivamente la función de pérdida. Con el objetivo de minimizar la función de pérdida, la actualización de los parámetros requiere el cálculo del gradiente de la función con respecto a las variables de pesos y *bias*, así:

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{m}\sum_{1}^{m}\frac{\partial l^{i}(\boldsymbol{w}, b)}{\partial \boldsymbol{w}} \tag{16}$$

$$b = b - \frac{\alpha}{m}\sum_{1}^{m}\frac{\partial l^{i}(\boldsymbol{w}, b)}{\partial b} \tag{17}$$

Donde, α representa un valor numérico conocido como tasa de aprendizaje. En estas dos últimas ecuaciones, el algoritmo de gradiente descendente opera sobre todo el conjunto de datos promediando su valor entre el número ejemplos (m).

Otro enfoque del algoritmo implica la conformación de un subconjunto de datos (conocido como minilote o minibatch), cada vez que se requiere actualizar los valores de los pesos y *bias*. Este enfoque se conoce como *Minibatch stochastic gradient descent.* En este caso, la actualización de los parámetros se realiza de la siguiente forma:

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial l^{i}(\boldsymbol{w}, b)}{\partial w} \tag{18}$$

$$b = b - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial l^{i}(\boldsymbol{w}, b)}{\partial b} \tag{19}$$

Donde *BS* representa el número de ejemplos de cada minilote (tamaño del lote o *batch size*) y α denota la tasa de aprendizaje.

Como paso previo a la aplicación del algoritmo por *Minibatch stochastic gradient descent,* es necesario inicializar los parámetros del modelo (w y b), lo cual por lo general se realiza de manera aleatoria. Posterior a esto, el algoritmo de forma iterativa toma minilotes aleatorios de los datos, y actualiza los parámetros en la

dirección del gradiente negativo. Tomando el gradiente de la función de pérdida con respecto a **w**, la actualización de los pesos se realiza de la siguiente forma:

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial\left(\frac{1}{2}\left(\hat{y}^i - y^i\right)^2\right)}{\partial w} \tag{20}$$

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial\left(\frac{1}{2}\left(\boldsymbol{w^T x^i} + b - y^i\right)^2\right)}{\partial w} \tag{21}$$

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{BS}\sum_{i\in BS}\boldsymbol{x^i}\left(\boldsymbol{w^T x^i} + b - y^i\right) \tag{22}$$

Por su parte, el valor de *bias* se actualiza restando al valor actual el término que incluye el gradiente de la función de pérdida respecto a b:

$$b = b - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial\left(\frac{1}{2}\left(\hat{y}^i - y^i\right)^2\right)}{\partial b} \tag{23}$$

$$b = b - \frac{\alpha}{BS}\sum_{i\in BS}\frac{\partial\left(\frac{1}{2}\left(\boldsymbol{w^T x^i} + b - y^i\right)^2\right)}{\partial b} \tag{24}$$

$$b = b - \frac{\alpha}{BS}\sum_{i\in BS}\left(\boldsymbol{w^T x^i} + b - y^i\right) \tag{25}$$

Como conclusión, el entrenamiento de los modelos consiste en realizar múltiples iteraciones sobre el conjunto de datos, tomando un minilote de ejemplos cada vez y actualizando los parámetros del modelo (*w, b*) en cada iteración. La iteración de todo el conjunto de datos se conoce como época, e involucra utilizar una vez cada lote del conjunto de datos de entrenamiento (siempre y cuando el número de ejemplos sea divisible por el tamaño del lote).

PARÁMETROS E HIPERPARÁMETROS

Durante el proceso de entrenamiento, el algoritmo de optimización permite actualizar de forma iterativa los parámetros del modelo, que para el modelo de regresión estudiado hasta aquí hemos denominado **w** y b. Sin embargo, el entrenamiento del modelo estipula valores adicionales como la tasa de aprendizaje, el tamaño del lote o el número de épocas, valores que típicamente no aprende el modelo durante el entrenamiento.

Estos valores que pueden ser ajustables pero que no se actualizan durante el entrenamiento se conocen como hiperparámetros. La selección de estos hiperparámetros se puede realizar con base en los resultados del entrenamiento, al evaluar el modelo sobre un conjunto de datos diferente al de entrenamiento que se conoce como conjunto de validación.

EJEMPLO DE IMPLEMENTACIÓN DE UN MODELO DE REGRESIÓN LINEAL

Para contextualizar un modelo de regresión lineal completo y aplicar los conceptos de este capítulo, será necesario considerar por los menos las siguientes fases:

- Datos de entrenamiento y lectura por lotes
- Creación del modelo
- Inicialización de parámetros
- Selección de función de pérdida
- Selección de algoritmo de optimización
- Entrenamiento del modelo

A continuación, se muestra el desarrollo de cada una de estas fases en un modelo de regresión lineal.

- Datos de entrenamiento y lectura por lotes

En este ejemplo se generarán 10000 datos de prueba que corresponden a las calificaciones obtenidas por 10000 estudiantes en una asignatura. Estas tres notas estarán entre 0.0 y 5.0, y por medio de ellas se generará la calificación

definitiva (etiqueta de salida) que se calcula considerando un 30% de la primera nota, 30% para la segunda nota y 40% para la tercera nota.

Lo anterior significa que los datos de entrenamiento se generarán de manera sintética, considerando tres atributos (x_1, x_2, x_3) con calificaciones entre 0.0 y 5.0. La nota definitiva será el valor a predecir, y para estos datos de entrenamiento se generan conociendo el peso de cada atributo, mediante la siguiente ecuación y código en Python:

$$y = 0.3x_1 + 0.3x_2 + 0.4x_3 \tag{26}$$

```
porcentajes = tf.constant([0.3, 0.3, 0.4])
b_ini = 0.0

atributos = tf.zeros((10000, 3))
atributos += tf.random.uniform(shape=atributos.sha
pe, minval=0.0, maxval=5.0)
etiqueta = tf.matmul(atributos, tf.reshape(porcent
ajes, (-1, 1))) + b_ini
etiqueta += tf.random.normal(shape=etiqueta.shape,
 stddev=0.2)
etiqueta = tf.reshape(etiqueta, (-1, 1))
```

En el código anterior, los pesos están definidos en la variable `porcentajes`, y no se incluye *bias* (b=0.0). La etiqueta de salida (`etiqueta`) se calculó utilizando la ecuación anterior, incluyendo cierto nivel de ruido en los datos mediante la adición de valores aleatorios obtenidos de una distribución normal con desviación estándar de 0.2.

Para el ejemplo, la dimensión de los atributos de entrada (en la variable `atributos`) tiene una dimensión de (10000, 3), mientras que la dimensión de la salida o etiqueta (variable `etiqueta`) es (10000, 1).

Para la lectura por minilotes, es preciso utilizar una función que permita crear un subconjunto de datos cuyos elementos sean extraídos de los tensores de entrada, conservando la estructura de estos. Esta operación se puede realizar por medio de la función `Dataset.from_tensor_slices` de tensorflow, junto con el método `batch` que permite combinar elementos consecutivos de un conjunto de

datos en lotes, de acuerdo con el tamaño del lote que se define al utilizar el método:

```
tam_lote=8

# Iteraciones para lectura del dataset en formato TensorFlow
dataset = tf.data.Dataset.from_tensor_slices((atributos, etiqueta))
dataset = dataset.shuffle(buffer_size=10000)
data_iter = dataset.batch(tam_lote)
```

Como buena práctica, también se han aleatorizado los datos de entrada, previo a la lectura por lotes (`dataset.shuffle`). A manera de prueba, se puede visualizar el contenido de una iteración, cuyo resultado serán 8 ejemplos del conjunto de datos, tanto para los atributos, como para la etiqueta:

```
next(iter(data_iter))
```

```
(<tf.Tensor: shape=(8, 3), dtype=float32, numpy=
 array([[2.2236247 , 1.8797708 , 3.2748823 ],
        [4.3598833 , 2.7310383 , 3.6872144 ],
        [1.310643  , 4.992275  , 4.745705  ],
        [4.1002517 , 3.413506  , 0.38547993],
        [4.4065337 , 1.3784158 , 1.4818233 ],
        [3.968305  , 0.14479339, 0.5596578 ],
        [4.7774534 , 2.0571613 , 2.067144  ],
        [3.0095048 , 2.2771108 , 3.656056  ]], dtype=float32)>,
 <tf.Tensor: shape=(8, 1), dtype=float32, numpy=
 array([[0.6800413],
        [2.6938043],
        [3.449986 ],
        [3.6250317],
        [1.6673617],
        [1.3380989],
        [1.7778645],
        [1.4929203]], dtype=float32)>)
```

En este caso, la respuesta muestra un tensor de dimensiones (8,3), lo que equivale a 8 ejemplos con 3 atributos cada uno. El segundo dato corresponde a un tensor de dimensiones (1,8), equivalente a la etiqueta de salida de cada uno de los 8 ejemplos.

- Creación del modelo

Como se ha comentado hasta aquí, el modelo de regresión lineal estudiado hasta aquí, que involucra la suma ponderada de los atributos de entrada (**X**) más el offset (b) para obtener el valor de salida estimado ($\hat{y}$) y se puede expresar mediante la Ecuación:

$$\hat{y} = \boldsymbol{X}\boldsymbol{w} + b \tag{27}$$

La implementación de este modelo en Python, se puede realizar mediante un producto punto entre los atributos de entrada (**X**) y los pesos del modelo (**w**) más el offset (b). Utilizando tensorflow, esta operación se puede realizar mediante la función `matmul` que permite multiplicar dos matrices. De esta forma, incluyendo el offset, el modelo puede definirse así:

```
def reg_lineal(X, w, b):
    return tf.matmul(X, w) + b
```

Sin embargo, este modelo también se puede formar mediante una relación como la mostrada en la siguiente gráfica (Saravia, 2021):

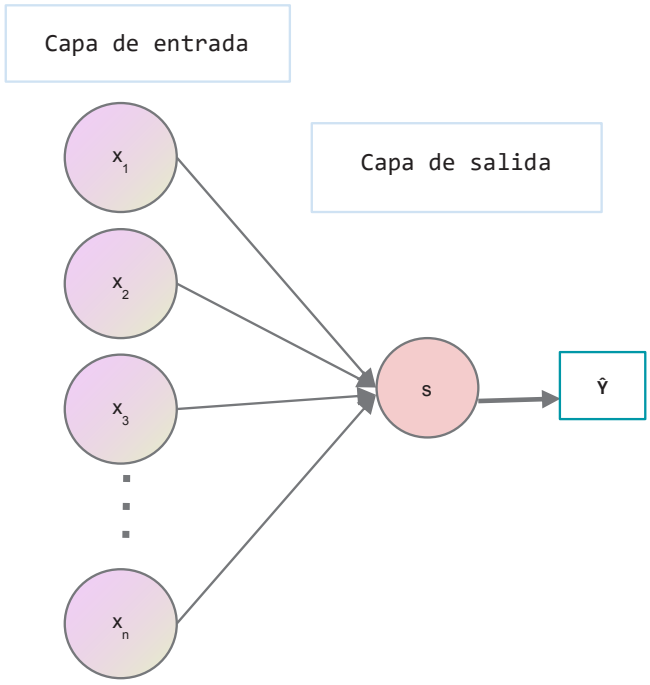

Figura 6. Ejemplo de regresión lineal mediante una red neuronal.

En la gráfica mostrada, los atributos de la entrada (x_1, x_2, …, x_n) están representados como una capa de entrada, cuyos valores aportan al cálculo del valor de salida con un peso determinado. Es decir, las líneas que unen cada atributo de entrada x con la salida, corresponden a los pesos de la regresión lineal. Al igual que en el modelo de regresión lineal, también se precisa de un valor de ajuste o sesgo (*offset* o *bias*).

El modelo mostrado corresponde a una red neuronal de una sola capa (la capa de salida, dado que la capa de entrada por lo general no se cuenta). Este tipo de capa en una red neuronal se conoce como densa o totalmente interconectada (*fully-connected (FC)*), dado que modela la relación entre las entradas y la salida como una suma ponderada y se implementa como una operación de multiplicación matriz/vector.

Para la implementación del modelo, se utilizará una API de alto nivel para `tensorflow`, conocida como `keras`. Las capas del modelo se enlazarán de manera secuencial, para lo cual `keras` dispone de la clase `Sequential` en la cual los datos de entrada se procesan en la primera capa, y su resultado será la entrada para la segunda capa y así sucesivamente.

Las capas tipo FC en keras se definen mediante la clase `Dense`, en la cual el argumento principal corresponde al número de neuronas o unidades de salida. Por su parte, el número de entradas de la capa `Dense` es calculado automáticamente por `keras` al ejecutar el modelo. De esta forma, al utilizar `tensorflow` para crear el modelo de regresión lineal quedaría así:

```
modelo = tf.keras.Sequential()
modelo.add(tf.keras.layers.Dense(1))
```

- Inicialización de parámetros

Dado que el algoritmo de optimización utilizado durante el aprendizaje involucra la actualización de los parámetros (**w**, b) a medida que avance el entrenamiento del algoritmo, es necesario definir un valor inicial previo a la primera iteración. Por lo general los pesos se inicializan con valores aleatorios. Este proceso en Python se puede realizar creando un tensor variable cuyos valores aleatorios se obtengan de una distribución normal. Asimismo, por lo general el *bias* se inicializa con cero, mediante un tensor variable:


```
w = tf.Variable(tf.random.normal(shape=(3, 1), mea
n=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)
```

Sin embargo, en la práctica, los valores iniciales se definen cuando el modelo se utiliza por primera vez (es decir que no es posible manipularlos desde el inicio). Para el caso específico del modelo con capa `Dense`, el método de inicialización se define como argumento de la capa. Para este ejemplo se utilizará el inicializador `RandomNormal`, que genera tensores con una distribución normal a partir de un valor de desviación estándar:

```
inicializador = tf.initializers.RandomNormal(stdde
v=0.01)
modelo_ini = tf.keras.Sequential()
modelo_ini.add(tf.keras.layers.Dense(1,kernel_init
ializer=inicializador))
```

Cabe hacer notar que tensorflow dispone de diversos algoritmos que han sido propuestos en la literatura para la inicialización de parámetros, cuyo diseño ha tenido en cuenta los requerimientos de los modelos actuales y que muchas veces ayudan a la convergencia del algoritmo, a la vez que evitan que parámetros diferentes converjan hacia un mismo valor, dado su carácter aleatorio. El listado de algoritmos de inicialización disponible en `tensorflow/keras` puede consultarse en la URL https://www.tensorflow.org/api_docs/python/tf/keras/initializers.

- Selección de función de pérdida

Tal como se comentó anteriormente, la función para calcular el error de un algoritmo en la predicción de los resultados puede programarse a partir de una métrica de distancia, como por ejemplo el error cuadrático:

```
Def perd_cuad(y_hat, y):
    return (y_hat - tf.reshape(y, y_hat.shape)) **
 2 / 2
```

Sin embargo, `keras/tensorflow` incluye el módulo `losses`, que incluye diversas clases y funciones para calcular el promedio de los errores. Por ejemplo,

la función `MeanSquaredError` permite calcular la media de los cuadrados de los errores entre las etiquetas y las predicciones. Su definición en tensorflow se puede realizar con una sola línea, así:

```
f_perdida = tf.keras.losses.MeanSquaredError()
```

El listado de funciones de pérdida disponible en `tensorflow/keras` puede consultarse en la URL https://www.tensorflow.org/api_docs/python/tf/keras/losses.

- Selección de algoritmo de optimización

La selección de un algoritmo de optimización como el *Minibatch stochastic gradient descent* comentado anteriormente, permite actualizar los valores de los parámetros con el fin de minimizar la función de pérdida. En este sentido, cuando se utilice el algoritmo de optimización durante el entrenamiento, los argumentos de entrada del algoritmo serán el conjunto de parámetros actual, la tasa de aprendizaje, y el tamaño del lote.

En este sentido, `Keras/tensorflow` dispone del módulo `optimizers`, que incluye diversas clases para realizar este proceso. Por ejemplo, la clase `SGD` permite disponer del optimizador de gradiente descendente (con lectura por minilotes, más un parámetro adicional conocido como impulso). Su selección en tensorflow se puede realizar con una sola línea, así:

```
optimizador = tf.keras.optimizers.SGD(learning_rat
e=0.002)
```

- Entrenamiento del modelo

Durante el entrenamiento del modelo, se realizarán iteraciones sobre el conjunto de datos (o subgrupos de este (minilotes)), donde un conjunto de iteraciones que involucre todo el conjunto de datos se conoce como época. Durante el entrenamiento, en cada una de estas iteraciones se realiza el siguiente procedimiento:

- Generar predicciones (mediante el modelo) y calcular la pérdida (proceso conocido como *forward propagation*).
- Calcular el valor de la pérdida y calcular los gradientes mediante el algoritmo (proceso conocido como *backpropagation*)
- Actualizar los parámetros del modelo mediante el optimizador

- Opcional: calcular y mostrar alguna métrica de rendimiento (para monitorear el progreso)

Como una aproximación de este proceso, se utilizará el siguiente bloque de código (Chollet, 2021; Zhang, 2021):

```
num_epochs = 4
for epoch in range(num_epochs):
    for X, y in data_iter:
        with tf.GradientTape() as tape:
            l = f_perdida(modelo_ini(X, training=T
rue), y)
        grads = tape.gradient(l, modelo_ini.traina
ble_variables)
        optimizador.apply_gradients(zip(grads, mod
elo_ini.trainable_variables))
    l = f_perdida(modelo_ini(atributos), etiqueta)
    print(f'época {epoch + 1}, pérdida {l:f}')
```

En este ejemplo se realizan cuatro lecturas completas del conjunto de datos (épocas), donde en cada época se realizan iteraciones del conjunto de datos con un tamaño de lote dado. Por ejemplo, si el conjunto de datos tiene 10000 datos, y se define 8 como el tamaño de lote, se tendrán 10000/8 = 1250 iteraciones en cada época. El método `GradientTape` utilizado en el código anterior facilita llevar un registro de los resultados relacionados con la diferenciación automática, cuyos gradientes serán utilizados por el optimizador para la actualización de los parámetros en cada iteración.

El resultado del entrenamiento, en este caso mostrará los valores de pérdida obtenidos para cada época:

```
época 1, pérdida 0.041337
época 2, pérdida 0.040648
época 3, pérdida 0.040433
época 4, pérdida 0.040402
```

Aunque para este ejemplo, se ha utilizado un bloque de código para el entrenamiento, el cual utiliza aspectos como la lectura por lotes, optimizador o la función de pérdida definidos previamente, *frameworks* como `tensforflow`

incluyen también soluciones para realizar el entrenamiento de los modelos. Esto se abarcará en más detalle en capítulos siguientes.

ALGORITMOS DE OPTIMIZACIÓN

Tal como se ha venido explicando, un algoritmo de optimización es una herramienta que permite la actualización recurrente de los parámetros de un modelo en aras de minimizar el valor de la función de pérdida (o función objetivo), cuando se evalúa en el conjunto de datos de entrenamiento. De aquí que, el rendimiento del algoritmo de optimización tiene una incidencia directa sobre la eficiencia del algoritmo de entrenamiento del modelo. Debido a esto, el estudio de este tipo de algoritmos y sus dependencias facilita un mejor ajuste de algunos hiperparámetros de un modelo.

En este sentido, hiperparámetros como la tasa de aprendizaje, la función de pérdida, el tamaño del lote o el número de épocas están directamente relacionados con el algoritmo de optimización, y cobran sentido una vez se comprende el principio de funcionamiento de dicho algoritmo.

Es importante tener en cuenta aquí que la función objetivo de un algoritmo de optimización corresponde a la función de pérdida. El diseño o selección de esta función debe estar orientada a medir el error de entrenamiento, entendido como la distancia entre el valor que predice un modelo respecto al valor real (obtenido a partir de las etiquetas de los datos de entrenamiento). Asimismo, el proceso de optimización debe propender por reducir el error de generalización del modelo, obtenido al evaluar un modelo sobre datos nuevos (datos que no fueron utilizados ni en entrenamiento ni en validación, pero que proceden de la misma distribución de datos).

Dado que el objetivo de algoritmo de optimización está orientado a minimizar la función de pérdida, este tipo de algoritmos presenta ciertos desafíos que dependen en gran medida del tipo de función objetivo. Particularmente, la búsqueda del mínimo de una función puede verse envuelta en mínimos locales, zonas de la señal con efecto silla o gradientes con tendencia a cero.

Los mínimos locales de una función se presentan cuando el valor de esta función objetivo en un punto dado es menor que los valores de dicha función en cualquier otro punto cercano (es decir, en una vecindad de dicho punto). En un determinado algoritmo de optimización, los mínimos locales podrían confundir a

un algoritmo de optimización, respecto al valor mínimo global que corresponde al mínimo de la función objetivo en todo el dominio.

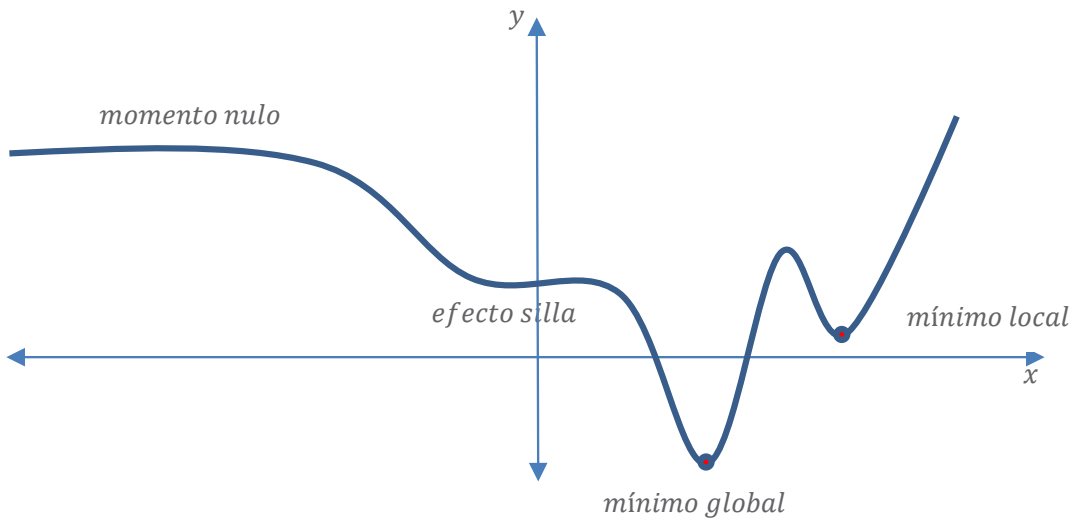

Figura 7. Mínimos de una función

La variación natural de los gradientes que se obtiene al utilizar pequeños subconjuntos del conjunto de datos de entrenamiento, como es el caso del algoritmo de gradiente descendente estocástico por minilotes (MSGD) facilita que el algoritmo sea capaz de sacar los parámetros de los mínimos locales.

En cuanto al efecto silla y gradientes nulos que pueden presentar algunas funciones, estos corresponden a segmentos de esta en los que el gradiente de la función parece atenuarse pero que no corresponden ni a un mínimo global ni a un mínimo local. De hecho, pueden existir zonas de una función (como por ejemplo aproximaciones asintóticas) en las cuales el algoritmo de optimización podría estancarse durante mucho tiempo.

Algoritmo de gradiente descendente

El gradiente descendente es un algoritmo de optimización que permite actualizar los parámetros de un modelo en la dirección que minimiza la función de pérdida, en la cual se involucra el cálculo del gradiente de la función ponderado por una constante conocida como tasa de aprendizaje.

Para entender por qué el algoritmo minimiza la función de pérdida, consideremos una función de costo unidimensional, *f(x)*, cuyo dominio y rango está definido por el conjunto de números reales. Dado que la función del algoritmo de optimización consiste en actualizar el parámetro del modelo (que para este ejemplo es *x*),

partamos de la expansión en serie de Taylor[9] de una función (infinitamente diferenciable)[10] alrededor de un punto $x=a$:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots, \tag{28}$$

Ahora consideremos un pequeño desplazamiento de la variable independiente, $x+\varepsilon$, y representemos la serie de Taylor en torno al punto x original. Es decir, utilizando la ecuación anterior reemplazamos *a* por *x*, y *x* en el término $(x-a)$ por $x+\varepsilon$:

$$f(x+\varepsilon) = f(x) + \varepsilon f'(x) + \varepsilon^2 \frac{f''(x)}{2} + \varepsilon^3 \frac{f^{(3)}(x)}{6} + \cdots, \tag{29}$$

Definiendo ahora Épsilon como el negativo del gradiente de la función multiplicado por una constante Alpha, es decir: $\varepsilon = -\alpha f'(x)$, se tiene:

$$f(x+\varepsilon) = f(x) - \alpha\big(f'(x)\big)^2 + \sum_{n=2}^{\infty} (-\alpha)^n \frac{f'(x) f^n(x)}{n!} \tag{30}$$

Si se utiliza una constante Alpha suficientemente pequeña, el tercer término del lado derecho de la ecuación se vuelve irrelevante. Además, si la derivada de la función es diferente de cero y la constante Alpha es positiva, el segundo término del lado derecho de la ecuación siempre será negativo. Esto implica que el valor de la función en el punto $(x+\varepsilon)$, es decir $f(x+\varepsilon)$, siempre será más pequeño que el valor de la función en el punto *x* (es decir, $f(x)$).

De acuerdo con lo anterior, con el objetivo de minimizar la función de costo, lo que busca el algoritmo de optimización de gradiente descendiente, es actualizar el parámetro o parámetros de la función de costo (es decir *x*), restándole al parámetro el gradiente de la función con respecto a dicho parámetro, es decir:

$$x = x - \alpha f'(x) \tag{31}$$

En términos generales, el algoritmo de gradiente descendente implica la selección tanto de un valor inicial para el parámetro *x* como el valor de una constante $\alpha>0$,

[9] https://en.wikipedia.org/wiki/Taylor_series
[10] https://mathworld.wolfram.com/TaylorSeries.html

para luego iterar continuamente el valor del parámetro *x* hasta una condición determinada (por ejemplo, el número de iteraciones o el número de épocas).

Un análisis similar al anterior se puede realizar para funciones multivariadas, llegando a la misma conclusión.

Gradiente descendente estocástico por minilotes

La aplicación del algoritmo de gradiente descendente puede considerarse desde dos enfoques, de acuerdo con el número de ejemplos del dataset utilizados en cada iteración del algoritmo de optimización. El primer enfoque (gradiente descendente) consiste en utilizar el conjunto de datos completo para calcular los gradientes y actualizar los parámetros. Sin embargo, el manejo de datos en este enfoque no es eficiente, dado que para una sola actualización de los parámetros es necesario leer y utilizar todo el dataset. Este aspecto puede tornarse de gran importancia al trabajar con conjuntos de datos muy grandes (por ejemplo, con millones de imágenes), lo que puede ralentizar el entrenamiento de un modelo.

En el otro extremo, el segundo enfoque (gradiente descendente estocástico) consiste en procesar cada vez un solo ejemplo del conjunto de datos, a partir de lo cual se actualizan los parámetros del modelo. Aunque este enfoque presenta un menor costo computacional en cada iteración comparado con el anterior, tampoco es eficiente desde el punto de vista de procesamiento de datos, dado que no aprovecha los recursos de paralelización y vectorización de los cuales disponen las unidades de procesamiento actuales (computacionales y gráficas).

En el medio de estos dos enfoques se encuentra el gradiente descendente estocástico por minilotes (*Minibatch stochastic gradient descent*), el cual puede ofrecer tanto un costo computacional bajo como una alta eficiencia estadística. Este enfoque consiste en leer subconjuntos o minilotes del conjunto de datos en lugar de ejemplos individuales ni tampoco leer todo el dataset. Una vez procesado el minilote, se calculan los gradientes y se actualizan los parámetros. De esta forma, el número de iteraciones que realiza el optimizador sobre el conjunto de datos está dado por:

$$\lceil DS/BS \rceil \tag{32}$$

donde *DS* corresponde al número de muestras del dataset completo, *BS* al tamaño (número de muestras) del minilote y el operador $\lceil\ \rceil$ corresponde a la función techo que devuelve el mínimo número entero no inferior al argumento.

Al utilizar el procesamiento por minilotes, el cálculo del gradiente se obtiene sobre el promedio de los datos:

$$\partial = \frac{\partial}{\partial \boldsymbol{w}}\left(\frac{1}{BS}\sum_{i\in BS} f(\boldsymbol{X}_i, \boldsymbol{W})\right) \tag{33}$$

Cuando se aplica el algoritmo de optimización por gradiente descendente estocástico por minilotes, el valor esperado del optimizador se mantiene sin cambios, mientras que la varianza se reduce significativamente (Zhang, 2021). En cuanto a tiempo de ejecución, en general la operación por minilotes es más rápida que la operación tanto por gradiente descendente estocástico como por gradiente descendente.

Ejemplo de algoritmo de optimización y tasa de aprendizaje

A manera de ejemplo, y como contexto de explicación del algoritmo de gradiente descendente, se tomará la siguiente función objetivo, que tiene su mínimo en *x=0*.

$$f(x) = \cosh\left({}^{\pi x}/_{4}\right) \tag{34}$$

Donde el cálculo del gradiente de *f(x)* da como resultado:

$$f'(x) = {}^{\pi}/_{4} * \sinh\left({}^{\pi x}/_{4}\right) \tag{35}$$

El cálculo de la función, sumado al cálculo del gradiente y respectiva actualización de *x* para 20 iteraciones arroja el siguiente resultado:

x	***α***	***f(x)***	***f'(x)***	***x-αf'(x)***
-1 (valor inicial)	0.05	1.3246	-0.6823	-0.9659 (valor siguiente)
-0.9659	0.5	1.3018	-0.6546	-0.6386
-0.6386	0.5	1.1284	-0.4106	-0.4333
-0.4333	0.5	1.0585	-0.2724	-0.2970
-0.2970	0.5	1.0273	-0.1849	-0.2046
-0.2046	0.5	1.0129	-0.1267	-0.1412
-0.1412	0.5	1.0062	-0.0873	-0.0976

-0.0976	0.5	1.0029	-0.0602	-0.0675
-0.0675	0.5	1.0014	-0.0416	-0.0466
-0.0466	0.5	1.0007	-0.0288	-0.0322
-0.0322	0.5	1.0003	-0.0199	-0.0223
-0.0223	0.5	1.0002	-0.0138	-0.0154
-0.0154	0.5	1.0001	-0.0095	-0.0107
-0.0107	0.5	1.0000	-0.0066	-0.0074
-0.0074	0.5	1.0000	-0.0046	-0.0051
-0.0051	0.5	1.0000	-0.0031	-0.0035
-0.0035	0.5	1.0000	-0.0022	-0.0024
-0.0024	0.5	1.0000	-0.0015	-0.0017
-0.0017	0.5	1.0000	-0.0010	-0.0012
-0.0012	0.5	1.0000	-0.0007	-0.0008

Tal como se aprecia en la tabla anterior, a medida que avanzan las iteraciones, el valor de *x* se aproxima gradualmente al valor en el cual la función objetivo es mínimo (*x=0*). Para comprobarlo, tomemos la función definida en Python:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

%matplotlib inline

# Función cosh y su gradiente
c = tf.constant(np.pi/4)
f = lambda x: tf.cosh(c * x)
gradf = lambda x: c * tf.sinh(c * x)
```

Como función de gradiente descendente, se utilizará la propuesta por (Zhang, 2021), donde es claro que, a partir del valor inicial, se actualiza el valor restando el gradiente de la función evaluada en ese punto.

```
def gd(alpha,valor_ini):
```



```
    x = valor_ini
    results = [x]
    for i in range(10):
        x -= alpha * gradf(x)
        results.append(float(x))
    return results
```

Los resultados del valor de *x* se almacenan en un vector para graficarlos sobre la función de costo, utilizando el siguiente código:

```
def grafica(valor_ini,parametros):
  eje_x = tf.range(valor_ini, -valor_ini, 0.01)
  y_curva = f(eje_x)
  y_puntos = f(parametros)

  # Plot the points using matplotlib
  plt.plot(eje_x, y_curva)
  plt.plot(parametros, y_puntos, marker = 'o', col
or = 'green')
  plt.xlabel('Argumento (x)')
  plt.ylabel('Función de costo')
  plt.title('Optimización')
  plt.legend(['Función de costo', 'Minimización it
erativa'])
  plt.show()
```

Para comprobar el comportamiento del algoritmo, se usará $x=-1$ como valor inicial, y $\alpha=0.05$ (tasa de aprendizaje).

```
valor_ini=-1
alpha=0.5
parametros = np.array(gd(alpha,valor_ini))
grafica(valor_ini,parametros)
```

De esta forma se obtiene el siguiente resultado, donde la curva azul corresponde a la función de costo a optimizar (*cosh*) y la curva verde corresponde a los valores de *x* para diferentes iteraciones. El número de valores de *x* mostrados en la gráfica es de 10, que corresponden a los primeros 10 valores de la tabla anterior.

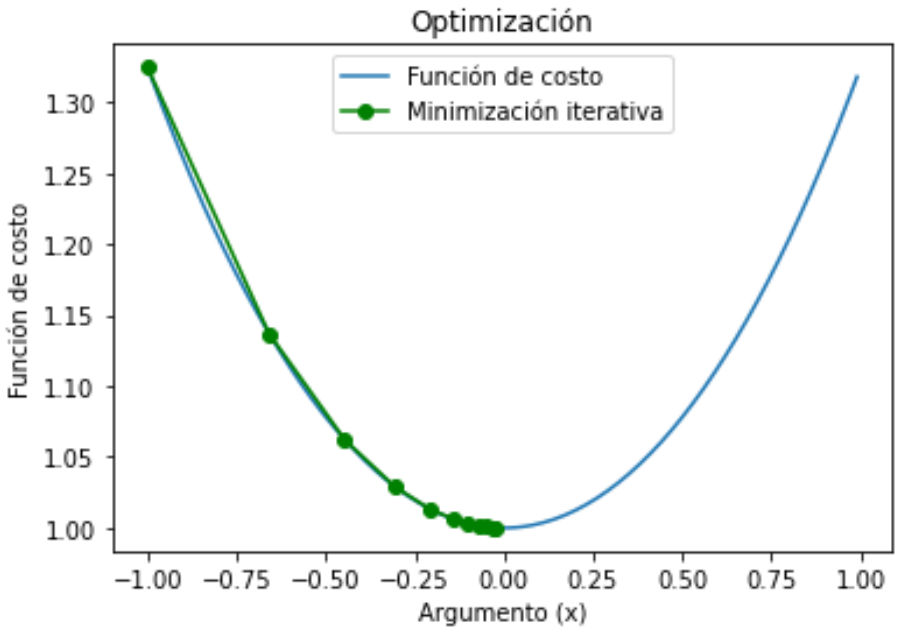

Figura 8. Ejemplo de algoritmo de optimización para función coseno hiperbólico y tasa de aprendizaje 0.5

A partir del valor inicial $x=-1$, el algoritmo de optimización muestra la aproximación del valor de x, hacia el punto mínimo de la función.

Además, al usar una tasa de aprendizaje muy pequeña puede hacer que la convergencia del algoritmo se ralentice, como se muestra a continuación:

```
valor_ini=-1
alpha=0.01
parametros = np.array(gd(alpha,valor_ini))
grafica(valor_ini,parametros)
```


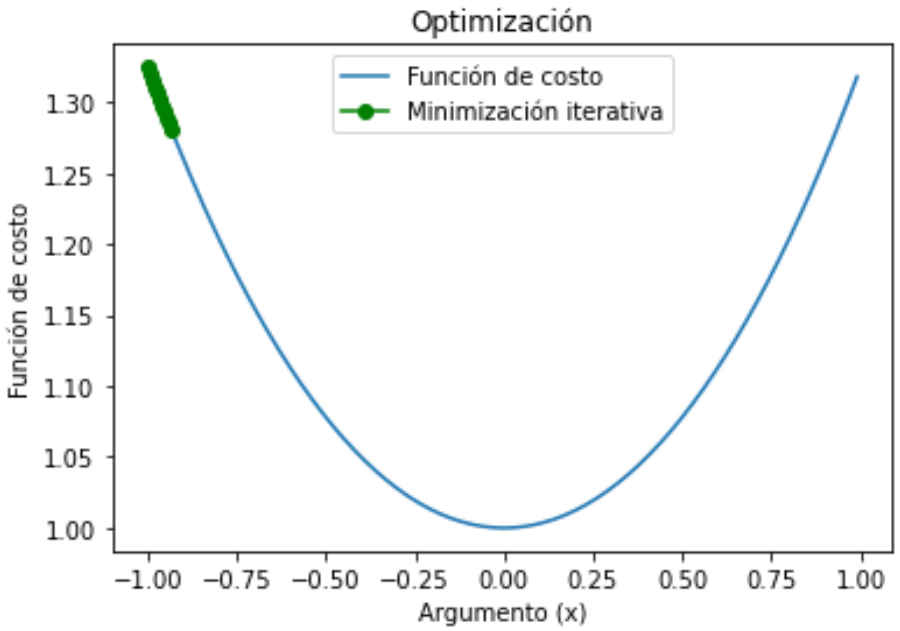

Figura 9. Ejemplo de algoritmo de optimización para función coseno hiperbólico y tasa de aprendizaje 0.01

Por el contrario, al usar una tasa de aprendizaje demasiado grande, no se garantiza que la iteración reduzca el valor de *x*. Para visualizar esto, se definirá una nueva función y su gradiente:

$$f(x) = {\sin(3\pi x)}/{x} \tag{36}$$

$$f'(x) = {(x\cos(3\pi x) - \sin(3\pi x))}/{x^2} \tag{37}$$

Para observar el comportamiento del algoritmo en esta nueva función, iniciaremos con una tasa de aprendizaje de 0.001 y un valor inicial de 0.1. Con estos valores, el comportamiento del algoritmo se aproxima gradualmente al mínimo global de la función:

```
c = tf.constant(3*np.pi)
f = lambda x: tf.sin(c * x)/x
gradf = lambda x: (x * tf.cos(c * x) - tf.sin(c *
x))/(x**2)

alpha=0.001
parametros = np.array(gd(alpha,0.1))
```

```
grafica(-2,parametros)
```

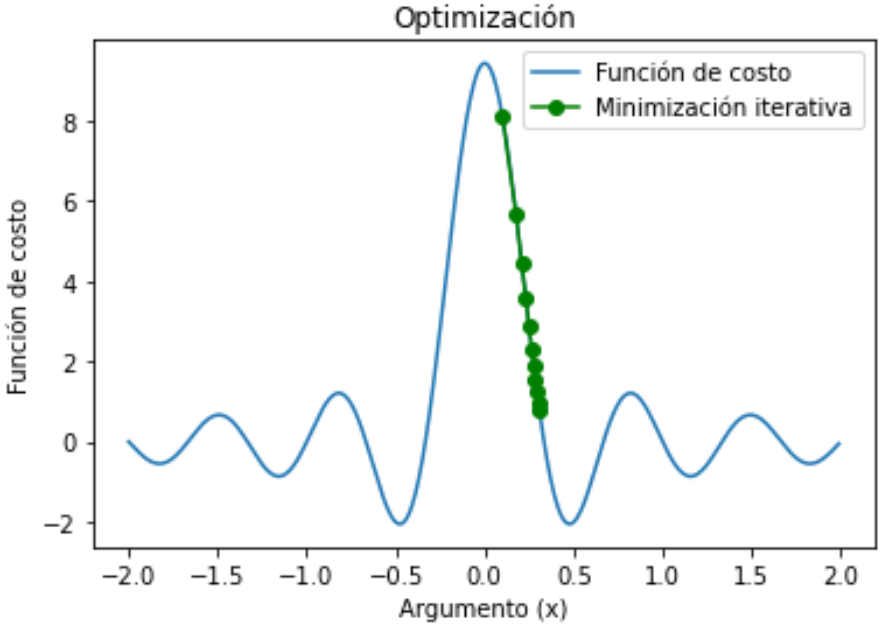

Figura 10. Ejemplo de algoritmo de optimización para función sinc y tasa de aprendizaje 0.001

Sin embargo, al aumentar significativamente el valor de la tasa de aprendizaje a 0.1 (e iniciar con un valor de 0.3), el algoritmo tendrá inconvenientes para reducir el valor del argumento:

```
alpha=1
parametros = np.array(gd(alpha,-1))
grafica(-4,parametros)
```


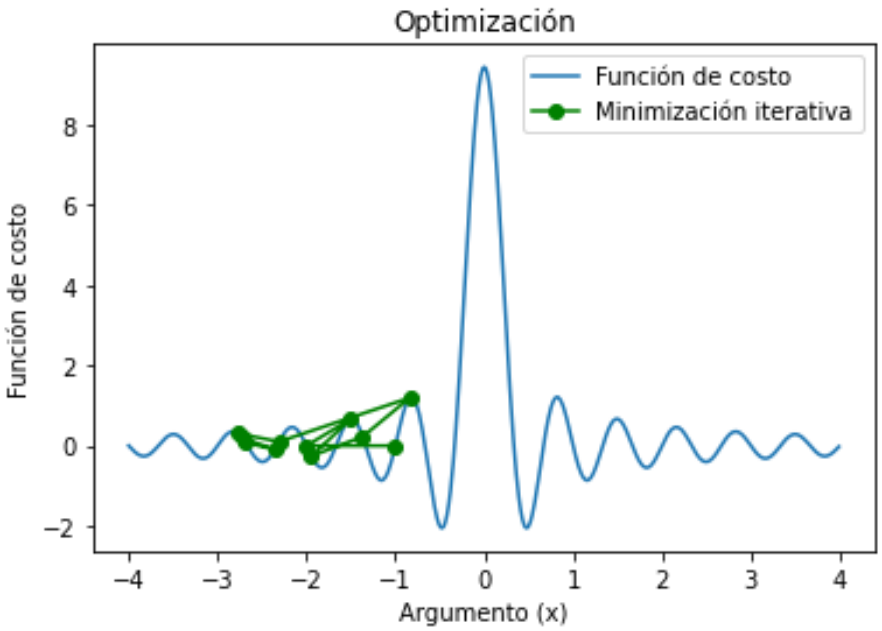

Figura 11. Ejemplo de algoritmo de optimización para función sinc y tasa de aprendizaje 1.0

Además, para una función que tenga muchos mínimos locales, el usar una tasa de aprendizaje grande, podría llevar también a una errónea localización del mínimo global:

```
alpha=0.1
parametros = np.array(gd(alpha,0.3))
grafica(-3,parametros)
```


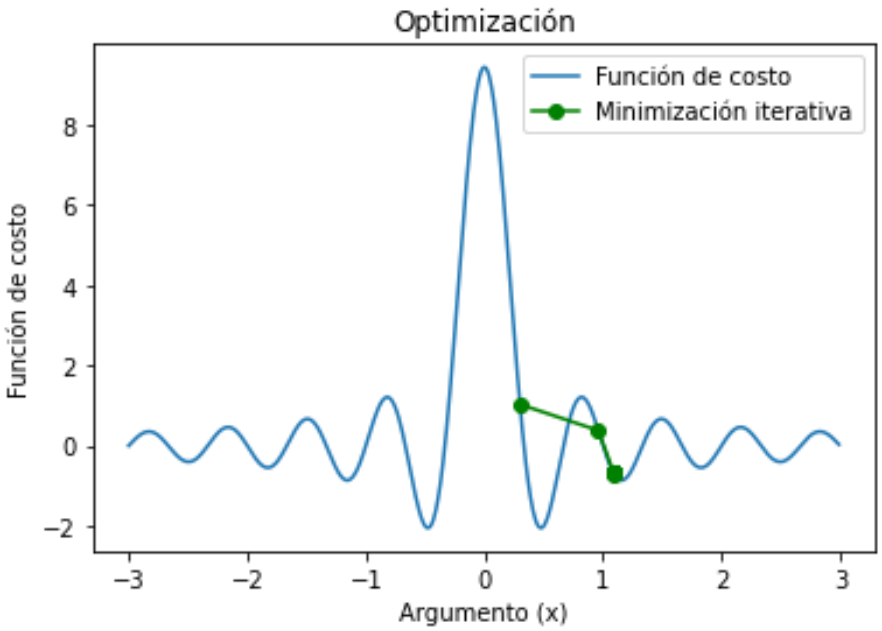

Figura 12. Ejemplo de algoritmo de optimización para función sinc y tasa de aprendizaje 0.1

Gradiente descendente estocástico

Como complemento al algoritmo de gradiente descendente, es importante considerar que normalmente la función objetivo se calcula como el promedio de la función de pérdida computada para todos y cada uno de los ejemplos del conjunto de datos de entrenamiento. Es decir,

$$f(x) = \frac{1}{m}\sum_{1}^{m} f_i(x) \tag{38}$$

De esta forma, el gradiente de la función objetivo se obtiene como el promedio del gradiente de la función de pérdida calculado para cada ejemplo del conjunto de datos de entrenamiento. Es decir:

$$\nabla f(x) = \frac{1}{m}\sum_{1}^{m} \nabla f_i(x) \tag{39}$$

El cálculo del gradiente como un promedio sobre el número de ejemplos, implica que el número de muestras es directamente proporcional al costo computacional del cálculo del gradiente. De aquí que el gradiente descendente estocástico (SGD) surge como una alternativa para reducir el costo computacional en cada iteración.

En particular, SGD propone en cada iteración seleccionar aleatoriamente un solo ejemplo del conjunto de datos mediante un muestreo uniforme. De esta forma, el costo computacional del cálculo del gradiente se reducirá, lo que es significativo para conjuntos de datos muy grandes. El precio a pagar por esta reducción involucra que la trayectoria de aproximación al mínimo de la función puede presentar mayores variaciones.

Tasa de aprendizaje dinámica

Como se comentó anteriormente, el tener una tasa de aprendizaje demasiado pequeña puede ralentizar el proceso de aprendizaje de tal manera que en cada iteración la aproximación hacia el mínimo de la función no sea significativo. Además, el tener una tasa de aprendizaje demasiado grande, por lo general no es una buena alternativa. Ante esta situación, el manejar una tasa de aprendizaje cuyo valor no sea constante para todas las iteraciones puede significar beneficios para el algoritmo de entrenamiento. Particularmente se puede iniciar con un valor determinado, e ir reduciendo su valor de acuerdo con una regla específica. De esta forma, es posible programar la tasa de aprendizaje para modular cómo cambia a lo largo del tiempo de entrenamiento. De esta forma, la sustitución de α por una tasa de aprendizaje variable en el tiempo, añade complejidad al control de la convergencia del algoritmo de optimización.

Actualmente, `Keras/tensorflow` dispone de las cuatro opciones listadas a continuación para programar la tasa de aprendizaje[11]:

- *Exponential Decay*: programa que aplica una función de decaimiento exponencial luego de cada paso del optimizador, de acuerdo con la siguiente Ecuación:

$$\alpha_i = \alpha_0 * r^{s_i/s} \tag{40}$$

Donde α_0 es la tasa de aprendizaje inicial, α_i es la tasa de aprendizaje en el paso i, r es la tasa de decaimiento, s_i es el número de paso y s es el número de pasos de decaimiento. De esta forma, la tasa de aprendizaje decrece gradualmente hasta alcanzar el r % de la tasa de aprendizaje original una vez transcurrido el número de pasos establecido.

[11] https://keras.io/api/optimizers/learning_rate_schedules/

Utilizando una tasa de aprendizaje inicial de 0.05, una tasa de decaimiento de 0.9 y 20000 pasos, el decaimiento exponencial se puede implementar de la siguiente manera:

```
alfa_inicial = 0.05
lr_schedule = tf.keras.optimizers.schedules.Exponenti
alDecay(
    alfa_inicial,
    num_pasos=20000,
    tasa_dec=0.9,
    staircase=True)
```

Así, la tasa de aprendizaje del optimizador se puede especificar como "`lr_schedule`" y no como un valor constante:

```
optimizer=tf.keras.optimizers.SGD(learning_rate=lr_sc
hedule)
```

- *Piecewise Constant Decay*: programa que calcula una tasa de aprendizaje constante por tramos luego del paso actual del optimizador, en función del número de paso actual y los límites de los tramos definidos, así:

$$\alpha = \begin{cases} \alpha[1] & cuando & s_i \leq límite[1] \\ \alpha[2] & cuando & límite[1] < s_i \leq límite[2] \\ \cdots & \cdots & \cdots \\ \alpha[n] & cuando & límite[n-1] < s_i \leq límite[n] \end{cases} \quad (41)$$

Donde $\alpha[j]$ es la tasa de aprendizaje definida para el tramo j, s_i es el número de paso y límite[k] corresponde al valor de paso que define la frontera de un determinado tramo. De esta forma, la tasa de aprendizaje para el primer tramo tendrá un valor de $\alpha[1]$ para el primer tramo, $\alpha[2]$ para el segundo tramo y así sucesivamente.

Con una tasa de aprendizaje inicial de 0.1 (en los primeros 1000 pasos), una tasa de aprendizaje de 0.01 en los siguientes 4000 pasos y una tasa de aprendizaje de 0.001 en los demás pasos, el decaimiento constante por tramos se puede utilizar de la siguiente manera:

```
step = tf.Variable(0, trainable=False)
limites = [1000, 5000]
tramos = [0.1, 0.01, 0.001]
learning_rate_fn = keras.optimizers.schedules.Piecewi
seConstantDecay(
```



```
    limites, tramos)
```

Así, la tasa de aprendizaje del optimizador se puede especificar en el optimizador:

```
learning_rate = learning_rate_fn(step)
optimizer=tf.keras.optimizers.SGD(learning_rate)
```

- *Polynomial Decay*: programa que aplica una función de decaimiento polinómico a un paso del optimizador, en función de la tasa de aprendizaje inicial, hasta alcanzar una tasa de aprendizaje final en un número determinado de pasos, de acuerdo con la siguiente Ecuación:

$$\alpha_i = \left((\alpha_0 - \alpha_{end})(1 - s_i/s)^p\right) + \alpha_{end} \quad (42)$$

Donde α_0 es la tasa de aprendizaje inicial, α_{end} es la tasa final de aprendizaje deseada, p es el orden del polinomio, s_i es el número de paso y s es el número de pasos de decaimiento. De esta forma, la tasa de aprendizaje decrece gradualmente hasta alcanzar el valor final una vez transcurrido el número de pasos establecido.

Utilizando una tasa de aprendizaje original de 0.05, una tasa final de decaimiento de 0.005 y 5000 pasos, el decaimiento polinómico se puede utilizar de la siguiente manera:

```
alfa_inicial = 0.05
alfa_final = 0.005
num_pasos = 5000
learning_rate_fn = tf.keras.optimizers.schedules.Poly
nomialDecay(
    alfa_inicial,
    num_pasos,
    alfa_final,
    power=0.5)
```

A partir de lo cual es posible especificar la tasa de aprendizaje del optimizador:

```
optimizer=tf.keras.optimizers.SGD(
                  learning_rate=learning_rate_fn)
```


- *Inverse Time Decay*: programa que aplica la función de decaimiento inverso a un paso del optimizador, en función de una tasa de aprendizaje inicial, de acuerdo con la siguiente Ecuación:

$$\alpha_i = \frac{\alpha_0}{(1 + r * s_i/s)} \tag{43}$$

Donde α_0 es la tasa de aprendizaje inicial, α_i es la tasa de aprendizaje en el paso i, r es la tasa de decaimiento, s_i es el número de paso y s es el número de pasos de decaimiento.

```
alfa_inicial = 0.01
num_pasos = 100
tasa_dec = 0.5
learning_rate_fn = keras.optimizers.schedules.Inverse
TimeDecay(
  alfa_inicial, num_pasos, tasa_dec)
```


