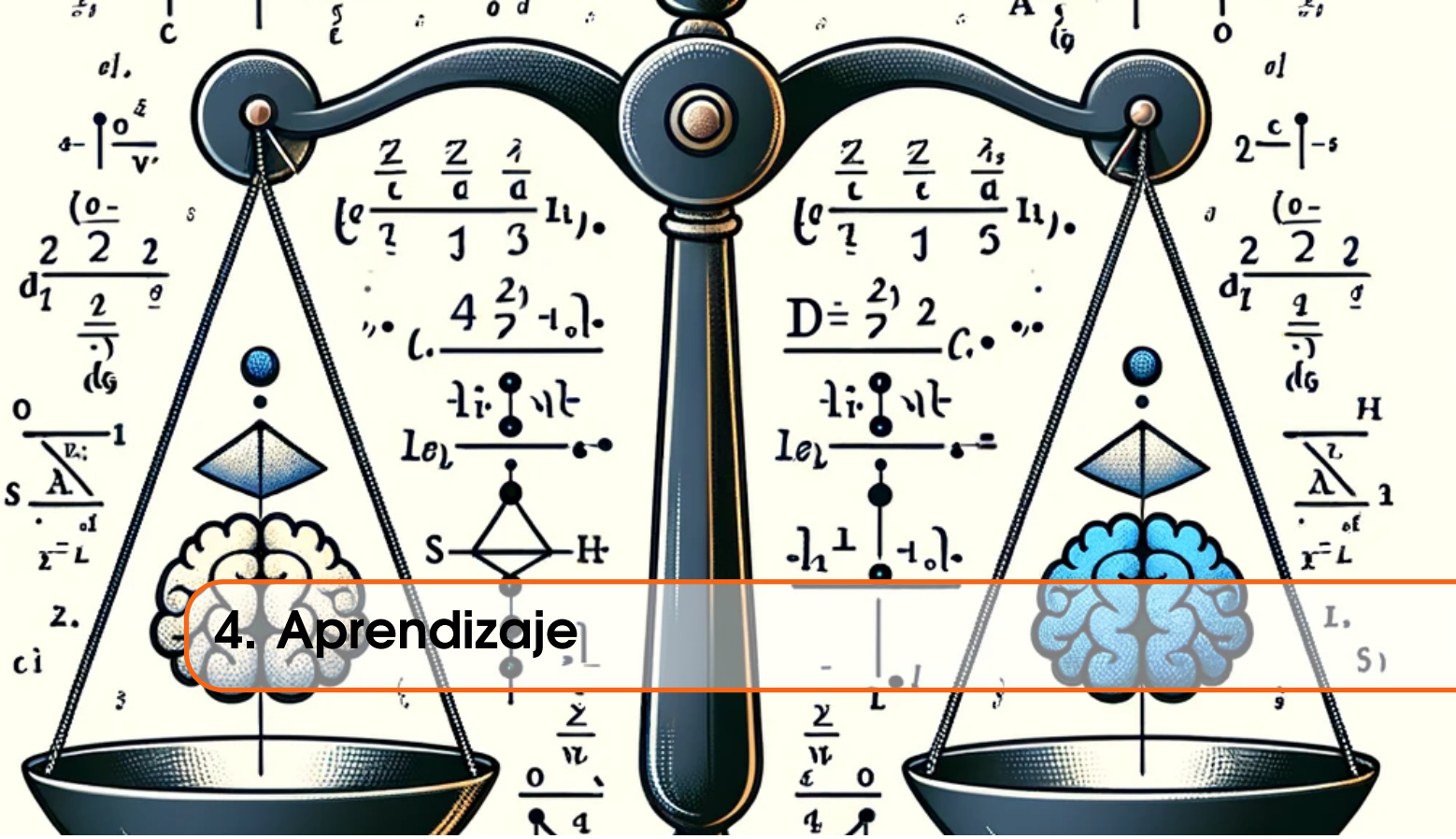


4. Aprendizaje

4.1 Introducción

El aprendizaje automático se basa en la idea de que las computadoras pueden aprender de manera similar a los humanos, mediante la observación y la experiencia. Para ello, el proceso de aprendizaje se divide en dos fases principales: entrenamiento y evaluación. Durante la fase de entrenamiento, el modelo aprende a partir de un conjunto de datos de entrenamiento, ajustando sus parámetros internos para minimizar la diferencia entre las predicciones del modelo y las etiquetas verdaderas de los datos.

Los algoritmos de aprendizaje hacen uso de algún tipo de optimización, que es la tarea de minimizar o maximizar una función $f(x)$ modificando a x. La función que buscamos optimizar es llamada **función objetivo**. Cuando se minimiza se le llama **función de costo**, **función de error**, o **función de perdida**.

4.2 Descenso por gradiente

Uno de los métodos más empleados para entrenar las redes neuronales debido a su efectividad y relativa eficiencia es el del descenso por gradiente. Imagina que estás en una montaña y tu objetivo es llegar al punto más bajo posible, pero está completamente oscuro y solo puedes sentir el terreno bajo tus pies. El algoritmo de descenso por gradiente es como tener un método para decidir en qué dirección dar el próximo paso basándote en la pendiente o inclinación del suelo que sientes con los pies. En vez de caminar al azar, decides moverte en la dirección en la que el terreno desciende más.

En este método la función objetivo está determinada por el error en las predicciones. $E(\theta)$, donde θ es un conjunto de parámetros. En la ecuación anterior no se especifica como parámetro las entradas de la red, x, ni los valores objetivo, y, porque durante el entrenamiento éstos se mantienen constantes.

4.2.1 Métricas de error

Existen diversas formas de medir el error, la más simple es usar el error residual de las predicciones:

$$e = (y - \hat{y}) \tag{4.1}$$

, donde y es el valor objetivo y $\hat{y}$ es la predicción. Este error es adecuado para una sola observación, sin embargo, no es fácil de generalizar a muchos valores dado que puede sufrir de cancelación o compensación de errores.

La suma de los cuadrados de los errores nos permite incorporar los errores de muchas predicciones en un solo valor escalar. La ecuación (4.2) muestra el SSE, donde el super-índice indica el número de ejemplo y m el total de ejemplos.

$$\texttt{SSE} = \sum_{i=1}^{m} (y^i - \hat{y}^i)^2 \tag{4.2}$$

El SSE contempla todos los errores sobre un conjunto de datos pero si los conjuntos de datos son muy grandes pueden causar un error de desbordamiento (*overflow*), que ocurre en computación cuando un cálculo produce un resultado que excede el rango máximo que puede ser representado con el número de bits asignados para ese tipo de datos en una computadora. Para solventar el problema se usa el error cuadrático promedio:

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^{m} (y^i - \hat{y}^i)^2, \tag{4.3}$$

donde m es el total de ejemplos en el conjunto de datos.

En muchos casos se le denomina también a la función de error como función de pérdida (*Loss function*). En este libro haremos referencia de forma indistinta a error y pérdida.

4.2.2 Proceso general de descenso por gradiente

Regresando al método de descenso por gradiente, a éste lo podemos sintetizar en los siguientes pasos:

1. Comenzamos con una conjetura inicial de los parámetros.

$$W = \texttt{sampling}() \tag{4.4}$$

2. Determinamos la dirección del mayor decremento de la función mediante el gradiente.

$$\nabla_W E = \frac{\partial E(W)}{\partial W} \tag{4.5}$$

3. Determinamos un cambio a partir de la dirección de decremento y un escalar (tasa de aprendizaje) que nos permite controlar que tanto se moverá la conjetura de los parámetros. Este cambio se suma a la conjetura previa de los pesos.

$$W = W - \eta \nabla_W E \tag{4.6}$$

4. Repetimos hasta encontrar un mínimo local.

Si el lector requiere más información acerca del método de descenso por gradiente general, éste puede consultar la sección 2.7 que describe el método e incluye ejemplos generales.

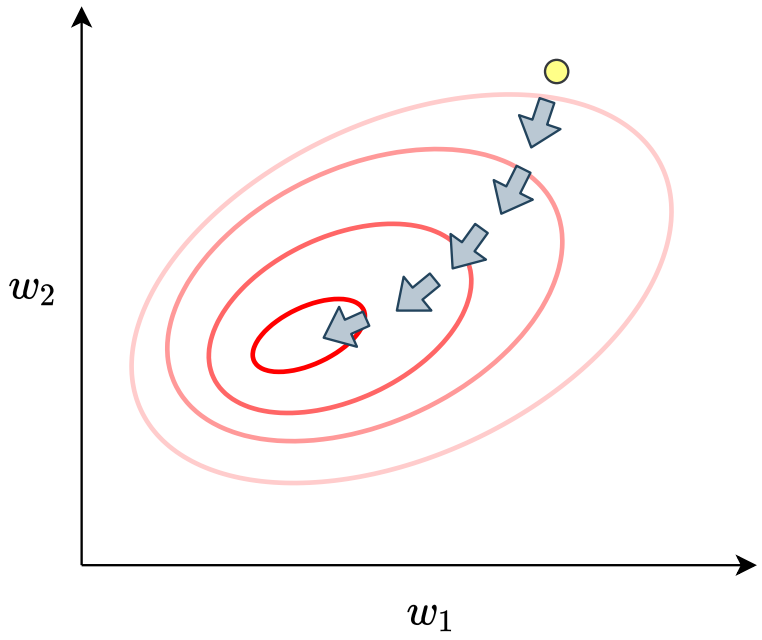


Figure 4.1: Ilustración del descenso por gradiente.

4.2.3 Uso en redes neuronales

La forma descrita previamente del descenso por gradiente a través de cuatro pasos es correcta, sin embargo, para poder llegar a una implementación práctica en las redes neuronales haremos algunas especificaciones y acomodos. Primero suponiendo el caso de una neurona simple y más tarde generalizándolo a neuronas con múltiples salidas.

En el primer paso del proceso, supondremos que el muestreo de los pesos se realiza de forma aleatoria bajo una distribución uniforme del espacio [-1, 1]. Más adelante abordaremos más formas de inicialización. Para el segundo paso, supondremos el caso de una neurona simple, es decir con una sola capa, y como función de error una versión modificada del MSE descrito en la ecuación 4.3:

$$E(\hat{y}) = \frac{1}{2m}\sum_{i=1}^{m}(y^i - \hat{y}^i)^2 \tag{4.7}$$

Por lo tanto la derivada del error con respecto de cada peso, w_i, se puede escribir como:

$$\nabla_{w_i} E = -(y - \hat{y}) f'(h) x_i \tag{4.8}$$

, donde $f'(h)$ indica la derivada de la función de activación. Reacomodando la ecuación (4.8) haremos la siguiente definición.

Definition 4.2.1 — Término de error. Se compone del producto entre el error residual y la derivada de la función de activación, es decir:

$$\delta = (y - \hat{y}) f'(h) \tag{4.9}$$

El término de error nos servirá como un indicativo de la dirección a la que nos tendremos que mover para minimizar la función de pérdida. Nótese que el término de error cambia el signo con respecto a la ecuación (4.8). Ésto se debe a que por definición el gradiente apunta en la dirección del cambio ascendente y el movimiento que requerimos debe ser en sentido contrario.

Para el tercer paso, primero calcularemos un **incremento de los pesos**, donde incluímos un nuevo hiperparámetro, η (eta), que controlará la magnitud de cambio, a esta variable se le conoce también como tasa de aprendizaje (*learning rate*).

Definition 4.2.2 — Incremento del peso. Se calcula a partir del producto entre la tasa de aprendizaje, el término de error y la entrada correspondiente. Es decir,

$$\Delta w_i = \eta \delta x_i \tag{4.10}$$

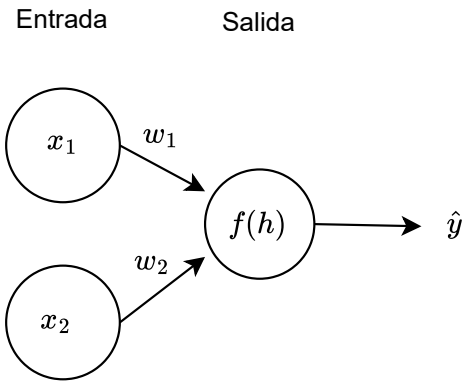
Finalmente, para completar la definición del descenso por gradiente se realiza la actualización de los pesos.

Definition 4.2.3 — Actualización del peso. El parámetro w_i se actualiza a partir del valor actual y el incremento del peso. Es decir,

$$w_i = w_i + \Delta w_i \tag{4.11}$$

El proceso de dividir el descenso por gradiente en tres partes, ecuaciones (4.9), (4.10) y (4.11) es importante desde el punto de vista computacional, dado que permite reutilizar el cálculo y volver la ejecución más eficiente.

■ **Example 4.1 — Descenso a un paso.** Suponga la siguiente configuración de una neurona simple:



donde $W = [0.4, 0.5]$, $b = -0.6$, $f = \sigma(h)$ y $X = [4, 2]$. Y sabiendo que la predicción actual de la red es $\hat{y} = 0.88$. Calcule los nuevos valores para los pesos a partir del descenso por gradiente a una época usando $\eta = 1$ y tomando en cuenta que el valor esperado es $y = 0.5$.

SOLUCIÓN: Siguiendo los pasos propuestos en la sección 4.2.3, calculamos el error MSE, E, con la ecuación 4.3:

$$E(\hat{y}) = \frac{1}{2m} \sum_{i=1}^{m} (y^i - \hat{y}^i)^2$$

$$E(\hat{y}) = \frac{1}{2}(0.5 - 0.88)^2 = 0.0722$$

Ahora calcularemos el término de error con la ecuación (4.9), pero primero calcularemos la derivada, f', de nuestra función sigmoide:

$$\sigma(h) = 0.88$$

$$f'(h) = \sigma(h)(1-\sigma(h)) = 0.06698$$

Entonces δ, será:

$$\delta = (y-\hat{y})f'(h) = (0.5-0.88)\cdot(0.06698) = -0.0221$$

El siguiente paso es calcular el incremento del peso y para eso usaremos la ecuación (4.10):

$$\Delta w_i = \eta\delta x_i, \eta = 1$$
$$\Delta w_1 = 1(-0.0221)(4) = -0.0884$$
$$\Delta w_2 = 1(-0.0221)(2) = -0.0442$$

Y para finalizar calcularemos los nuevos pesos con la ecuación (4.11):

$$w_1 = w_1 + \Delta w_1 = 0.4 + (-0.0884) = 0.3116$$
$$w_2 = w_2 + \Delta w_2 = 0.5 + (-0.0442) = 0.4558$$

Siendo nuestro vector de pesos actualizados

$$W = [0.3116, 0.4558]$$

■

■ **Example 4.2 — Descenso por varias épocas.** Continúe con el ejemplo 4.1 y calcule los valores de los pesos para tres épocas de tal forma que se complete la siguiente tabla.

Época	$\hat{y}$	Residual	Δ_W	Nuevo W
1	0.880	-0.330	[-0.088,-0.044]	[0.311,0.455]
2	0.826	-0.326	[-0.187,-0.093]	[0.124,0.362]
3	0.650	-0.150	[-0.136,-0.068]	[-0.012,0.243]
4	0.459	0.040	[0.04, 0.02]	[0.027, 0.263]

■

4.2.4 Actualización para múltiples ejemplos

En los últimos años, las redes neuronales han logrado avances impresionantes en una amplia gama de aplicaciones gracias a la disponibilidad de conjuntos de datos a gran escala. Estos conjuntos de datos proporcionan el "combustible" necesario para entrenar modelos profundos y complejos, lo que ha permitido avances significativos en áreas como visión por computadora, procesamiento del lenguaje natural, y el análisis de voz.

Entre los conjuntos de datos que se usan actualmente están lo siguientes. ImageNet [2]: que se utiliza para tareas de clasificación de imágenes. Actualmente, contiene más de 14 millones de imágenes distribuidas en 1,000 categorías. COCO (Common Objects in Context) [9]: un conjunto de datos ampliamente utilizado para tareas de reconocimiento de objetos y segmentación; contiene más de 330,000 imágenes, con más de 1.5 millones de instancias de objetos etiquetados. MassiveText [15]: utilizado en el preentrenamiento de modelos de lenguaje, contiene múltiples petabytes de datos textuales extraídos de diversas fuentes, incluidos sitios web, repositorios de código, y literatura técnica. Entre otros.

Por lo anterior, es necesario que el método de entrenamiento contemple la contribución que tiene cada ejemplo a las predicciones de la red y en consecuencia a la pérdida. Esto se logra a partir de extender el incremento del peso definido en la ecuación 4.11 a múltiples ejemplos.

Suponiendo un total de m tuplas de ejemplos, $(x^\mu, y^\mu) \in Data$, el incremento ahora se calcula considerando las contribuciones de cada ejemplo al error y promediando las contribuciones por el

número de ejemplos. Es decir, para el peso w_i que conecta la entrada x_i su incremento se calcula como:

$$\Delta w_i = \frac{1}{m} \sum_{\mu=1}^{m} \eta \, \delta^{\mu} x_i, \tag{4.12}$$

donde δ^{μ} es el término de error para la μ-ésima predicción:

$$\delta^{\mu} = (y^{\mu} - \hat{y}^{\mu}) f'(h^{\mu}) \tag{4.13}$$

4.2.5 Implementación eficiente

Si bien las definiciones para el incremento y actualización de los pesos son correctas, la implementación de dichas ecuaciones se puede eficientar a través de reutilizar las cálculos. Por lo tanto, una posible implementación se presenta en el algoritmo 15. Dicha implementación requiere el conjunto de datos, el número de registros, la tasa de aprendizaje y el número de épocas como entrada. Y entrega el conjunto de pesos actualizado. Como podemos observar en el algoritmo, líneas 6 y 8, el incremento de los pesos se hace reutilizando el término de error que es calculado para cada ejemplo. De forma similar, en la línea 8 no se incluye la tasa de aprendizaje ni el promedio (definidos en la ecuación (4.12)) debido a que mueven a la actualización en la linea 12. Esto tiene como consecuencia que solo se calcule una vez dicha multiplicación. En pocos ejemplos este cambio no tiene efecto, sin embargo, para miles o millones de ejemplos con miles de épocas de ejecución el ahorro será significativo.

Data: Conjunto de datos (*Data*), número de registros (m), tasa de aprendizaje (η), número de épocas (*epochs*)
Result: Pesos entrenados (W)

1 `Inicializar`(W); `/* Inicializar pesos */`
2 **for** $e = 1; epochs$ **do**
3 Ceros(ΔW_i) ;
4 **foreach** $(x, y) \in Data$ **do**
5 $\hat{y} = f(h(x, W))$; `/* Pase frontal */`
6 $\delta = (y - \hat{y}) f'(h)$; `/* Término de error */`
7 **foreach** $w_i \in W$ **do**
8 $\Delta w_i = \Delta w_i + \delta x_i$; `/* Incremento */`
9 **end**
10 **end**
11 **foreach** $w_i \in W$ **do**
12 $w_i = w_i + \frac{\eta}{m} \Delta w_i$; `/* Actualización normalizada */`
13 **end**
14 **end**
15 **return** W

Algorithm 1: Descenso por gradiente para una neurona simple usando un conjunto de datos de longitud arbitraria.

Con respecto a la codificación del método en Python. El código del cuadro 4.1 implementa el entrenamiento de una red neuronal simple utilizando el método de descenso por gradiente. Comienza definiendo el número de épocas (*epochs*) y la tasa de aprendizaje (*learnrate*).

Luego, en un bucle *for* que recorre cada época de entrenamiento, se inicializa un incremento en los pesos ($incremento_w$) como una matriz de ceros del mismo tamaño que los pesos originales.

```
for e in range(epochs):
  incremento_w = np.zeros(weights.shape)
```

Dentro de este bucle principal, hay otro bucle que itera sobre cada par de características (*features*) y objetivos (*targets*). La función *zip* permite emparejar las características con los objetivos.

```
for x, y in zip(features.values, targets):
```

Para cada par, calcula la salida de la red neuronal (output) aplicando la función de activación sigmoidal a la suma ponderada de las características y los pesos actuales.

```
h = np.dot(x, weights)
output = sigmoid(h)
```

Luego, se calcula el término de error en dos instrucciones: calcular el error residual y multiplicar por la derivada de la función de activación,

```
error = y - output
delta = error*sigmoid_prime(h)
```

A continuación se acumula el incremento. Hay que tener en cuenta que la entrada x es un vector y al usar el operador * se escala el vector por el término de error.

```
incremento_w += delta * x
```

Después de procesar todos los datos de entrenamiento, se normaliza el incremento de los pesos dividiendo por el número total de muestras (m), y luego actualiza los pesos sumándoles el producto del incremento en los pesos y la tasa de aprendizaje.

```
m = len(features.values)
weights += (learnrate/m) * incremento_w
```

En resumen, este código entrena una red neuronal durante un número específico de épocas, ajustando los pesos de acuerdo con el gradiente descendente para minimizar el error entre las predicciones del modelo y los valores objetivo.

4.2.6 Ejercicios

Exercise 4.1 Muestre que la derivada del error con respecto de los pesos para la suma de los errores cuadráticos es igual al término de error. ■

Exercise 4.2 Suponiendo que la red del ejercicio 3.5-1 deba tener una salida objetivo $y = 0.95$:

1. Determine el error residual, e, de la red.
2. Determine el término de error, δ.
3. Determine el incremento de los pesos Δw_i usando una tasa de aprendizaje $\eta = 10$.
4. Actualice el valor de cada w_i.

■

4.3 Retropropagación

La retropropagación es un método para calcular el gradiente necesario para ajustar los pesos de una red multicapa. Este método se incrusta dentro del descenso por gradiente y lo habilita para entrenar las redes multicapa. En su forma teórica, el método utiliza el error de la capa de salida y propaga este error hacia atrás, capa a capa hasta la capa de entrada.

```
def descenso_por_gradiente(ejemplos, valores_objetivo, tasa_de_aprendizaje,
                                        epocas):
    # Variables varias
    n_ejemplos, n_caracteristicas = ejemplos.shape
    Historial_error = []

    # Inicializacion de pesos
    pesos = np.random.normal(scale=1 / n_caracteristicas**.5, size=
                                          n_caracteristicas)

    for e in range(epocas):
        incremento_w = np.zeros(pesos.shape)
        for x, y in zip(ejemplos.values, valores_objetivo):
            # Inferencia
            h = np.dot(x, pesos)
            salida = sigmoid(h)

            # Calculo de error
            error = y - salida

            # Termino de error
            delta = error*sigmoid_prime(h)

            # Acumulacion del gradiente
            incremento_w += delta * x

        # Actualizacion de pesos
        m = len(ejemplos.values)
        pesos += (tasa_de_aprendizaje/m) * incremento_w

        # Registro de error
        out = sigmoid(np.dot(ejemplos, pesos))
        error = np.mean((out - valores_objetivo) ** 2)
        Historial_error.append(error)
        if e % (epocas / 10) == 0:
            print("Epoca:", e, " Error: {:.3f}".format(error))

    return pesos, Historial_error
```

Table 4.1: Implementación en python del descenso por gradiente para una red simple.

Siendo este método una extensión del descenso por gradiente. Partimos del hecho que los pesos se actualizan de la misma forma iterativa, siendo el incremento del peso proporcional a la tasa de aprendizaje y al gradiente del error. A diferencia de la ecuación () definida previamente, en este caso, la el gradiente se calcula con respecto al peso independientemente de la capa. Observe la ecuación (4.14).

$$w^{new} = w^{old} + (-1)\eta\nabla_w E \tag{4.14}$$

Calcular el gradiente implica determinar las derivadas parciales con respecto del peso en cuestión. Supongamos que se desea calcular el gradiente para un peso (i, j), de la capa t; escrito como $w_{t,i,j}$. El peso dentro de la red se ilustra en la figura 4.2.

$$H_t = A_{t-1}W_t + B \tag{4.15}$$

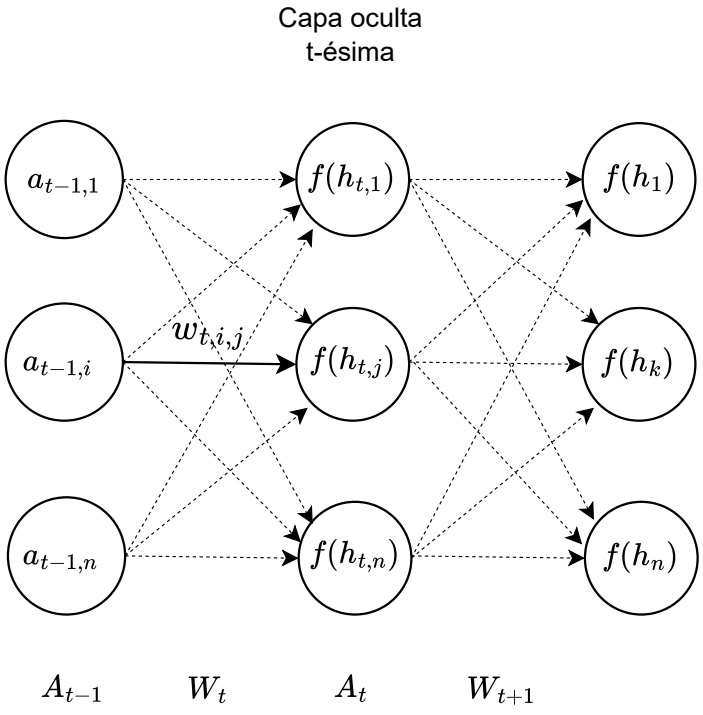


Figure 4.2: Ilustración del peso $w_{t,i,j}$. El peso conecta la i-ésima neurona de la capa $t-1$ con la neurona j-ésima de la capa t.

$$A_t = f(H_t) \tag{4.16}$$

El gradiente para este peso se determina usando la regla de la cadena. Acortando la escritura de la derivada escribiríamos:

$$\frac{\partial E}{\partial w_{t,i,j}} = \frac{\partial E}{\partial h_{t,i}} \frac{\partial h_{t,j}}{\partial w_{t,i,j}}, \tag{4.17}$$

donde el término $\frac{\partial E}{\partial h_{t,i}}$ está indicado y se tendría que desplegar de acuerdo con la regla de la cadena. Por otro lado, usando la expansión de la ecuación (4.15) podríamos resolver el segundo término.

$$\frac{\partial h_{t,j}}{\partial w_{t,i,j}} = \frac{\partial}{\partial w_{t,i,j}} \left(\sum_k a_{t-1,k} \cdot w_{t,k,j} + b_{t,j} \right) \tag{4.18}$$

Dado que tenemos una derivada parcial, el único término donde no se hace cero la derivada es donde k iguala a i, por tanto:

$$\frac{\partial h_{t,j}}{\partial w_{t,i,j}} = a_{t-1} \tag{4.19}$$

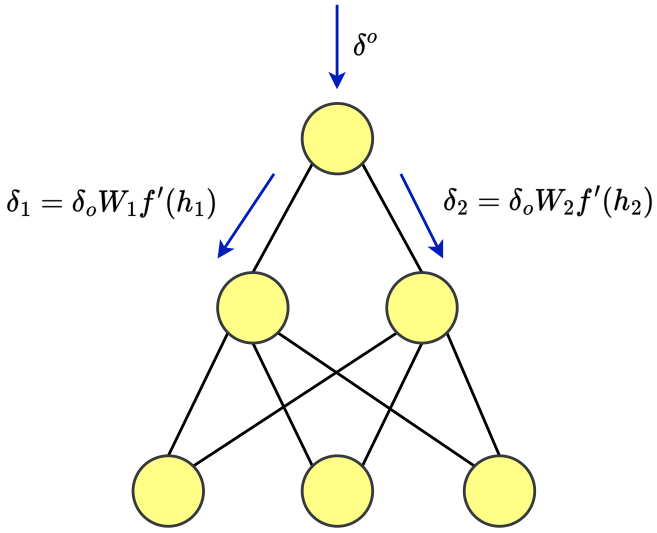


Figure 4.3: Ilustración de la relación del término de error con las capas superiores.

Reemplazando en la ecuación (4.17),

$$\frac{\partial E}{\partial w_{t,i,j}} = \frac{\partial E}{\partial h_{t,i}} a_{t-1}, \tag{4.20}$$

De forma general, el peso $w_{t,i,j}$ se actualizaría como:

$$w_{t,i,j} = w_{t,i,j} + (-1)\eta \frac{\partial E}{\partial h_{t,i}} a_{t-1} \tag{4.21}$$

De lo anterior podemos observar nuevamente que se requiere un proceso iterativo que actualizará al peso. Además que se tendrá que calcular en cada iteración la derivada del error con respecto del valor h. Si bien el método es correcto, estudiaremos en la siguiente sección una reconfiguración para evitar cálculos repetitivos y mejorar la eficiencia de la retropropagación.

4.3.1 Retropropagación eficiente

Como vimos en la sección previa, la retropropagación nos permite calcular la contribución de cada peso al error, y con base en ello ajustar su valor. En esta sección estudiaremos una forma de mejorar la eficiencia del cálculo del gradiente de forma independiente. Al final, tendremos una forma recurrente de calcular el gradiente, a través de un término de error similar al que calculamos en la sección 4.2.3. Esto nos permitirá reusar el término de error inmediato superior e independizar el cálculo de capas superiores.

Caso base

Iniciaremos con el análisis de la retropropagación en la capa superior. Esta comienza comienza calculando el error de la capa de salida, especificada con el sub-índice **o**. Observe la figura 4.3. De acuerdo con lo especificado en la sección 4.2.3, el término de error se determina como:

$$\delta_o = \frac{\partial E}{\partial \hat{y}} \frac{\partial f}{\partial \hat{h}} = (y - \hat{y}) f'(h) \tag{4.22}$$

El término δ_o nos permite actualizar la capa de salida. Si embargo si deseamos actualizar la capa oculta, especificada con el sub-índice h. Debemos calcular su contribución del peso esto es:

$$\frac{\partial E}{\partial w_h} = \frac{\partial E}{\partial \hat{y}} \frac{\partial f_o}{\partial h_o} \frac{\partial h_o}{\partial a_h} \frac{\partial f_h}{\partial h_h} \frac{\partial h_h}{\partial w_h} \tag{4.23}$$

Substituyendo la ecuación 4.22 y resolviendo $\frac{\partial h_o}{\partial a_h}$ reescribimos el gradiente como:

$$\frac{\partial E}{\partial w_h} = \delta_o w_o \frac{\partial f_h}{\partial h_h} \frac{\partial h_h}{\partial w_h} \tag{4.24}$$

$\frac{\partial f_h}{\partial h_h}$ indica la derivada parcial de la función de activación oculta, dado que la función puede cambiar en cada implementación, la dejaremos indicada como: $f_h'(h_h)$ y reescribimos la ecuación como:

$$\frac{\partial E}{\partial w_h} = \delta_o w_o f_h'(h_h) \frac{\partial h_h}{\partial w_h} \tag{4.25}$$

Lo anterior nos permite definir un término de error en la capa oculta que depende de las capas superiores:

$$\bar{\delta_h} = \delta_o w_h f'(h_h) \tag{4.26}$$

Observemos que este nuevo término de error es proporcional al error en la capa de salida, δ_o, y a la conexión que se tiene con dicha capa, w_h.

En la mayoría de los casos, las redes tienen más de una salida por lo cual, generalizaremos que el término de error en la capa oculta se calcula como:

$$\delta_h = \sum_o w_o \delta_o f'(h_h) \tag{4.27}$$

Una vez establecido el término de error en la capa oculta, reutilizaremos los conceptos de incremento y actualización. Donde el incremento de los pesos se calcula de la misma forma que en la ecuación (4.10):

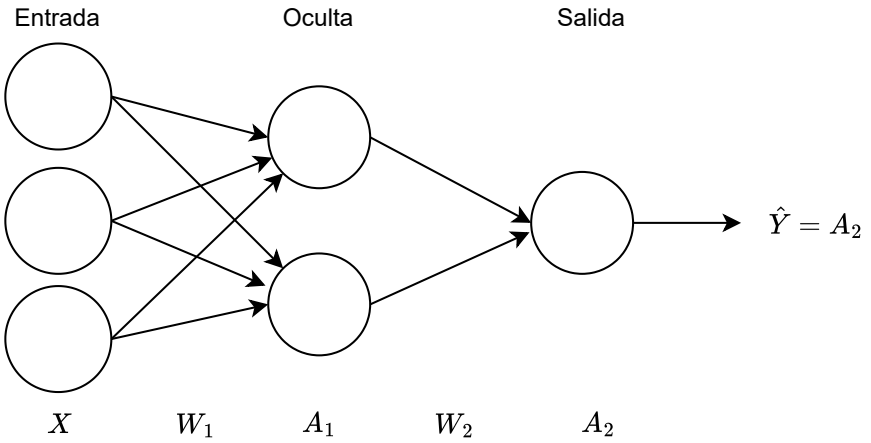
$$\Delta w_{h,i} = \eta \delta_h x_i \tag{4.28}$$

Caso general

$$\delta_{i,j} = \sum_k w_{i+1,k} \delta_{i+1,k} f_i'(h_{i,j}) \tag{4.29}$$

$$\Delta W_{i,j} = \eta \cdot \delta_{i,j} \cdot A_{i-1} \tag{4.30}$$

■ **Example 4.3** Partiendo de la siguiente red de una capa oculta:



■

4.3.2 Algoritmo de retropropagación

Data: Conjunto de datos (*Data*), número de registros (m), tasa de aprendizaje (η), número de épocas ($epochs$)
Result: Pesos entrenados ($\mathbf{W}$)

```
Inicializar(W);                                          /* Inicializar pesos */
for e = 1; epochs do
    Ceros(ΔW) ;
    foreach (X, y) ∈ Data do
        ŷ = Φ_W(X) ;                                     /* Pase frontal */
        δ_t = (y − ŷ) f'(h) ;                            /* Capa de salida */
        for i = 1; t − 1 do
            δ_{i,j} = Σ_k δ_{i+1,k} w_{i+1,k,j} f'(h_{i,j}) ;
        end
        foreach ΔW_i ∈ W do
            foreach ΔW_{i,j} ∈ ΔW_i do
                ΔW_{i,j} = ΔW_{i,j} + δ_{i,j} · A_{i−1,j} ;   /* Acumulación del incremento */
            end
        end
    end
    foreach W_i ∈ W do
        W_i = W_i + (η/m) ΔW_i ;                         /* Actualización matricial */
    end
end
return W
```

Algorithm 2: Algoritmo de retropropagación.

4.3.3 Implementación de la retropropagación

4.4 Inicialización de los pesos

No todos los métodos para encontrar los pesos, conocidos como algoritmos de optimización, siguen el mismo enfoque. Hay algoritmos que, directamente, buscan una solución sin necesidad de hacer iteraciones. Estos son ideales, pero no siempre son aplicables a redes neuronales. Por otro lado, muchos de los algoritmos que sí usamos en aprendizaje profundo como el descenso por gradiente requieren de un proceso iterativo, es decir, avanzan paso a paso hacia la solución. Esto implica que debes definir un punto de partida para estas iteraciones, lo que no es una tarea menor.

El punto de inicio en el entrenamiento de una red neuronal es crucial. Dependiendo de dónde comiences, el algoritmo puede encontrar una buena solución eficientemente o, por el contrario, puede que ni siquiera logre encontrarla. Algunos puntos de partida pueden hacer que el proceso de aprendizaje sea tan inestable que el algoritmo se tope con problemas numéricos y no funcione en absoluto. Si el algoritmo logra converger, el punto de inicio también influirá en cuán rápido lo hace y si el resultado es óptimo o no en términos de rendimiento y coste. Incluso soluciones con costes similares pueden comportarse de manera muy diferente en situaciones reales, mostrando errores de generalización dispares. Por tanto, la elección del punto de inicio no solo afecta la posibilidad de aprender algo sino que tan bién se aprende.

De acuerdo con [6] los métodos actuales de inicialización son simples y heurísticas dado que no se conoce complementamente el área de optimización de redes neuronales.

4.4.1 Problema del mal condicionamiento

La inicialización de los pesos en una red neuronal es el proceso mediante el cual se asignan valores iniciales a las conexiones entre las neuronas de las diferentes capas de la red. Uno de los problemas que se busca evitar al inicializar los pesos de forma adecuada es el problema de mal condicionamiento, que se refiere a la ineficacia del método de optimización para alcanzar su objetivo. Este problema puede surgir si los valores iniciales de los pesos son demasiado grandes o demasiado pequeños. Si los pesos iniciales son muy grandes, esto puede llevar a que los gradientes (los valores que se calculan durante la retropropagación para ajustar los pesos) sean demasiado grandes, lo que puede hacer que el proceso de entrenamiento sea inestable y que los pesos crezcan exponencialmente; además es posible que se sufra también de error de overflow que indica que la computadora no es capaz de representar dichas cantidades. Por el contrario, si los pesos iniciales son muy pequeños, los gradientes también serán pequeños, lo que ralentizará el aprendizaje y puede hacer que el entrenamiento se estanque.

Otra razón importante para una buena inicialización es evitar la explosión o desaparición del gradiente. Esto es particularmente problemático en redes profundas, donde los gradientes pueden volverse extremadamente grandes o pequeños a medida que se propagan hacia atrás a través de las capas, lo que dificulta el aprendizaje. Por ejemplo, una mala inicialización puede provocar que las neuronas en capas anteriores aprendan mucho más lentamente que las de capas posteriores, lo que hace que la red en su conjunto sea difícil de entrenar.

Como ejemplo, observemos el comportamiento de la función sigmoide que fue muy utilizada en las primeras propuestas de redes. Conforme el gradiente se aleja del centro su evaluación es cero. Esto dificulta el entrenamiento cuando las predicciones son de igual forma lejanas al cero. Ocasionando un desvanecimiento del gradiente, sobre todo cuando la misma función de activación es usada en varias capas.

4.4.2 Inicializaciones prácticas

Existen varias formas de inicializar los pesos en una red neuronal, y cada una está diseñada para mejorar la convergencia y evitar problemas comunes como la desaparición o explosión del gradiente.

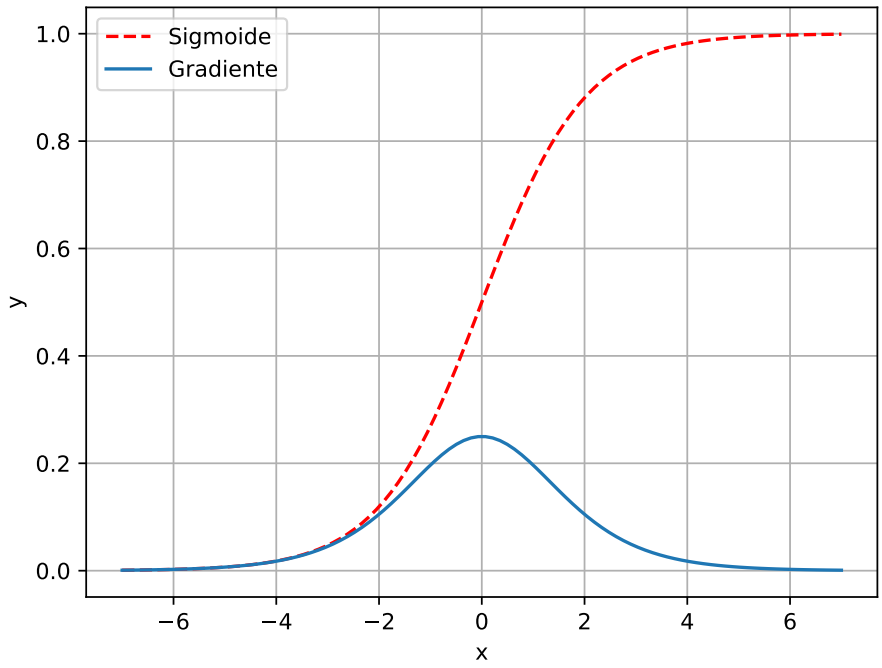


Figure 4.4: Función de activación sigmoide y su gradiente.

A continuación, se describen las principales técnicas de inicialización de pesos:

Inicialización aleatoria uniforme

Esta técnica asigna valores aleatorios a los pesos dentro de un rango específico de valores uniformemente distribuidos. El rango puede ser simétrico alrededor de cero, por ejemplo, entre -0.1 y 0.1.

$$W \sim \mathscr{U}(-a, a) \tag{4.31}$$

donde a es un valor que delimita el rango.

En numpy la inicialización aleatoria uniformse se puede implementar como:

```
np.random.uniform(low=-a, high=a, size=(dim_out, dim_in))
```

Inicialización Gaussiana o normal

En esta estrategia, los pesos se inicializan con valores aleatorios que siguen una distribución normal (gaussiana) con media cero y una desviación estándar pequeña.

$$W \sim \mathscr{N}(0, \sigma) \tag{4.32}$$

donde σ es la varianza.

Inicialización de Xavier

Propuesta por Xavier Glorot y Yoshua Bengio [5], esta técnica fue diseñada para redes con activaciones sigmoides o tangente hiperbólica. La idea es establecer los pesos de tal manera que

las entradas y salidas de cada capa tengan varianzas similares, lo que ayuda a mantener el flujo de los gradientes a lo largo de la red. Los pesos se inicializan de acuerdo a una distribución uniforme o normal, pero el rango de los valores se escala en función del número de neuronas de entrada y salida de una capa.

$$W \sim \mathscr{U}\left(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right) \tag{4.33}$$

donde n es el número de neuronas en la capa anterior.

Inicialización de He

Esta inicialización fue propuesta por Kaiming He [7] y está diseñada para redes con activaciones ReLU (Rectified Linear Units) o variantes como Leaky ReLU. La inicialización de He ajusta los valores de los pesos para que las activaciones no se saturen ni se apaguen, lo cual es crucial para entrenar redes profundas con activaciones ReLU. Los pesos se inicializan según una distribución normal escalada por el número de neuronas de entrada de cada capa.

$$W \sim \mathscr{N}\left(0, \frac{2}{n_{\text{in}}}\right) \tag{4.34}$$

donde n_{in} es el número de neuronas de entrada.

$$W_{i,j} \sim U(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}) \tag{4.35}$$

where $U(-a,a)$ is the uniform distribution in the interval $(-a,a)$ and n is the number of input units.

4.5 Sobre-ajuste

El sobre-ajuste o *overfitting* ocurre cuando la red neuronal se ajusta en sobre-medida a los datos de entrenamiento. Cuando esto sucede, la red no funciona con precisión ante nuevos ejemplos; es decir, el modelo no puede generalizar los nuevos datos consecuencia de que ha aprendido tan bien los datos de entrenamiento. Por ejemplo, una red podría ser sobre-ajustada a reconocer perros de raza fench puddle; en consecuencia no va a poder reconocer otras razas, aunque pertenezcan a la misma clase "perro", dado que las otras razas tienen características similares pero no exactamente iguales.

La directa de evitar sobre-ajuste es aumentando la diversidad de datos. Al emplear más ejemplos en el conjunto de entrenamiento es posible alimentar al modelo con información relevante para algunos otros casos que puedan presentarse; sin embargo, si estos datos no están limpios y solamente aportan ruido causarán que le tome al modelo mucho más tiempo encontrar una solución. Además, en la mayoría de los casos conseguir nuevos datos suele ser costoso en términos de tiempo y recursos económicos.

La segunda estrategia a utilizar para combatir el sobre ajuste es la aumentación de datos. En esta estrategia se suele agregar ruido o transformaciones geométricas a los datos disponibles. Por ejemplo, en el caso de imágenes se pueden aplicar transformaciones fotométricas, de perspectiva, de escala o de recortes aleatorios. En todos los casos de aumentación se debe vigilar que a pesar de las transformaciones el objeto de estudio mantenga propiedades que puede encontrarse en otros ejemplos de la misma clase.

Por el contrario, cuando ya no es posible la adquisición de datos ya sea originales o sintéticos, se suelen utilizar otras técnicas denominadas regularización. Éstas tienen por objetivo modificar el proceso de aprendizaje con el fin de disminuir el sobre-ajuste pero manteniendo los datos disponibles.

4.6 Regularización

La regularización es una técnica que hace pequeñas modificaciones al modelo o el entrenamiento buscando que el primero generalice mejor. A continuación estudiaremos algunas alternativas.

4.6.1 Paro anticipado

El paro anticipado (*early stopping*) es un método que busca detener el entrenamiento antes de que el modelo comience a sobre-ajustarse a los datos utilizados en esta etapa. Una desventaja de este método es que no es posible asegurar que el entrenamiento se detenga en el mínimo que se puede alcanzar durante el entrenamiento.

4.6.2 Regularización L1 y L2

Estas estrategias de regularización actualizan las funciones de costo adicionando un término llamado "término de regularización". La adición de este término permite que los valores de las matrices de pesos sean reducidas, por que asume que una red neuronal con matrices de pesos pequeñas conduce a modelos más simples; además de reducir completamente el sobre ajuste.

L1

El modelo regresión Lasso o mejor conocido como L1 es una forma de regularización, que se escribe de la siguiente forma:

$$\Omega(\theta) = \lambda \parallel w \parallel_1 = \lambda \sum_i |w_i| \tag{4.36}$$

que es la suma de los valores absolutos de los pesos individuales, siendo λ el parámetro de regularización. Y al agregarlo a la función de costo, lo podemos escribir como:

$$F_{costo} = F_{error} + \lambda \sum_i |w_i| \tag{4.37}$$

L2

Conocida como regresión de Ridge, decaimiento de peso o regularización L2, adiciona una penalidad de normas de la forma:

$$\Omega(\theta) = \frac{\lambda}{2} \parallel w \parallel_2^2 \tag{4.38}$$

donde de la misma forma λ es el parámetro de regularización. Y la función de costo se reescribe como:

$$F_{costo} = F_{error} + \frac{\lambda}{2} \parallel w \parallel_2^2 \tag{4.39}$$

4.6.3 Deserción

La deserción o *dropout* provee un método de regularización computacionalmente no costoso pero poderoso, este método puede ser visto como una forma de hacer empaquetamiento de conjuntos de ensambles de redes neuronales muy grandes. El empaquetamiento implica el entrenamiento de múltiples modelos y evaluar múltiples modelos para cada dato de prueba. El calculo de *dropout* se realiza mediante una distribución probabilista $p^{(i)}(y|x)$.

4.7 Optimización

La optimización es una rama de las matemáticas que se centra en encontrar el mejor resultado o solución de entre un conjunto de alternativas posibles bajo ciertas restricciones o condiciones. Se utiliza en diversos campos para maximizar o minimizar funciones objetivo. En matemáticas, estas funciones pueden representar costos, errores, eficiencia, rendimiento, entre otros aspectos, y las restricciones pueden ser limitaciones físicas, recursos, o reglas que deben cumplirse.

En el contexto de las redes neuronales y el aprendizaje profundo, la optimización juega un papel crucial ya que se utiliza para ajustar los parámetros de los modelos de redes neuronales. El objetivo aquí es minimizar la función de pérdida, que mide qué tan bien el modelo predice el resultado esperado durante el entrenamiento. Al reducir el valor de esta función de pérdida, el modelo se hace más preciso y eficaz en sus predicciones.

El proceso de optimización en el aprendizaje profundo generalmente se lleva a cabo mediante algoritmos de optimización, como el descenso del gradiente y sus variantes (por ejemplo, el descenso del gradiente estocástico, Adam, RMSprop). Estos algoritmos ajustan iterativamente los pesos de las conexiones en la red neuronal basándose en el gradiente de la función de pérdida con respecto a los pesos. Al hacer estos ajustes, el modelo aprende de los datos de entrenamiento.

En esta sección retomaremos el camino iniciado en la sección del descenso por gradiente y abordaremos diversas variantes que han obtenido resultados positivos en el entrenamiento de redes con decedas o centenas de capas.

4.7.1 Descenso por gradiente estocástico (SGD)

El descenso por gradiente estocástico es una extensión del descenso por gradiente, que como ya hemos visto anteriormente el objetivo de este método es descender hasta llegar a un valor mínimo de una función objetivo y de esta forma reducir el error que pueda tener una red neuronal al momento de realizar predicciones sobre un conjunto de datos.

Aquí el término estocástico hace referencia a aleatorio, esto es porque el SGD toma un pequeño conjunto de datos (*minibatch*) del conjunto de datos disponible para el entrenamiento, esta estrategia es útil en casos en los que se tiene bases de datos enormes y se recomienda utilizarlo cuando el cuello de botella es el tiempo de entrenamiento disponible ya que esta estrategia es mucho más rápida en términos temporales.

La imagen 4.5 nos muestra el comportamiento del SGD, en donde observamos como desde el punto de inicialización se comienza a descender aunque de una forma que parece “zigzag" o con desplazamientos no tan directos como se puede observar cuando se emplea todo el conjunto de datos.

4.7.2 Momento (*momentum*)

El momento es un término que se le puede ver como una especie de memoria que almacena cual es la dirección que tiene el gradiente, tratando de conservar la dirección que presenta el gradiente y lo podemos ver de la siguiente forma:

$$M_t = \beta M_{t-1} + (1-\beta)\nabla_w \mathscr{L}(W,X,y) \tag{4.40}$$

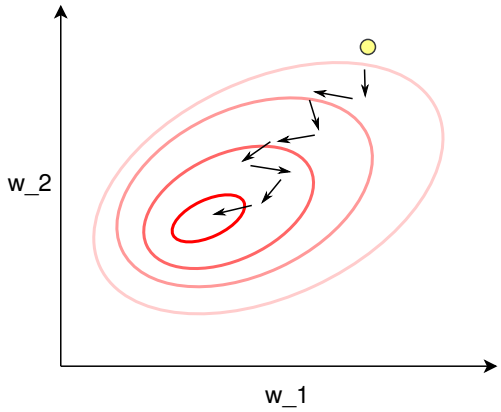


Figure 4.5: Gráfico del comportamiento del SGD

y la actualización de los pesos como:

$$W_t = W_{t-1} - \eta M_t \tag{4.41}$$

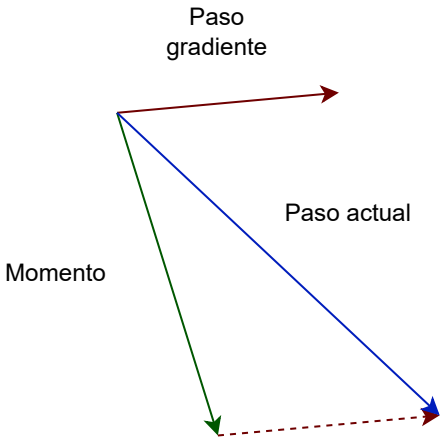


Figure 4.6: El momento en SGD

En la imagen 4.6 se puede ver que se tiene un vector del momento que tiene una dirección, al calcular el gradiente se tiene el vector de paso de gradiente que apunta a una dirección diferente, finalmente el paso que se dará es resultado de la combinación de ambos vectores; esto porque se busca priorizar la dirección que se tiene previamente y como se observa en 4.7 el trayecto que toma ahora el SGD ya no luce “erratico", es un descenso que va siendo marcado por todas las flechas negras pero corregido por el momento.

4.7.3 Caída de la tasa de aprendizaje *Learning rate decay*

Sabemos que establecemos el paso que podemos dar durante el entrenamiento, pero una vez que se logra llegar a la región cercana al mínimo es posible que sea necesario reducir este paso, es decir realizar movimientos más pequeños para mantenerse en la región cercana al mínimo y esta actualización está dada por:

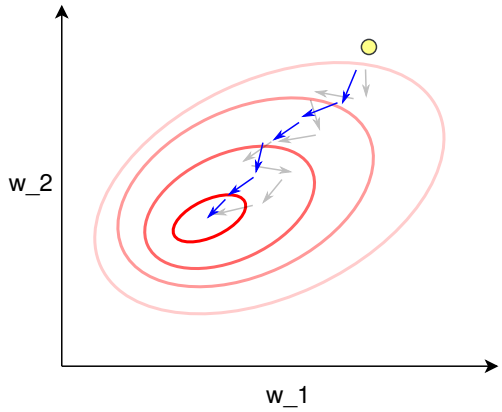


Figure 4.7: Gráfico del comportamiento del SGD con momento

$$\eta = \eta_0 \times \frac{1}{1 + i\mathscr{D}} \tag{4.42}$$

donde η_0 es la tasa de aprendizaje inicial, $\mathscr{D}$ es la caída e i es la época de entrenamiento; esto da como resultado a una caída de la tasa de aprendizaje con respecto a la época de entrenamiento en la que se encuentra la red (como se puede observar en 4.8).

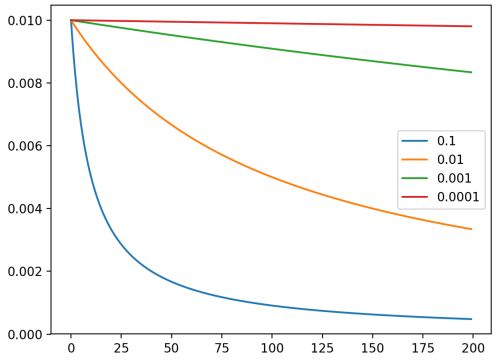


Figure 4.8: Efecto de la caída ($\mathscr{D}$) sobre la tasa de aprendizaje. Se observa la caída que presentan sobre distintos valores con respecto a la época en la que se encuentra.

4.8 Normalización

4.9 Ejercicios

Exercise 4.3 Partiendo del ejercicio 3.8 calcule:

- El vector de términos de error, δ
- La matriz de incrementos, ΔW
- Actualice los pesos.

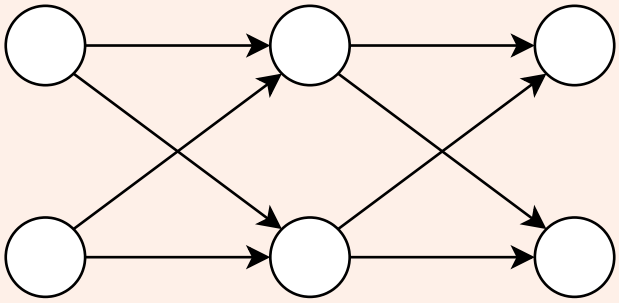
■

Exercise 4.4 Partiendo del ejercicio 3.10, suponga una métrica de pérdida como el MSE, $y = 0.5$, $\eta = 10$. Determine lo siguiente:

- Determine el error residual.
- Determine los términos de error.
- Determine las matrices de actualización.
- Calcule los nuevos valores de todos los pesos.

■

Exercise 4.5 Dada la siguiente red neuronal multicapa:



donde:

$$X = [6,7], W^1 = \begin{bmatrix} .4 & .2 \\ .8 & 1.0 \end{bmatrix}, W^2 = \begin{bmatrix} .1 & .8 \\ .4 & .3 \end{bmatrix}, Y = [1.0, 0.5],$$

funciones de activación *sigmoides*, error cuadrático medio como pérdida, $\eta = 1.0$, y sin sesgos.

1. Calcule la salida vectorial de la red.
2. Determine el vector de error residual.
3. Determine el término de error de salida, δ^o.
4. Determine el término de error de la capa oculta δ^h
5. Calcule las matrices de actualización, Δ^1 y Δ^2
6. Actualice el valor de los pesos.

■

4.10 Acordeón de aprendizaje

Errores:

$$e = (y - \hat{y})$$

$$\text{SSE} = \sum_{i=1}^{m} (y^i - \hat{y}^i)^2$$

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^{m} (y^i - \hat{y}^i)^2$$

Derivadas de las funciones de activación:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

$$\tanh'(x) = 1 - \tanh^2(x)$$

Término de error en la capa oculta:

$$\delta_h = \sum_o w_o \delta_o f'(h_h)$$