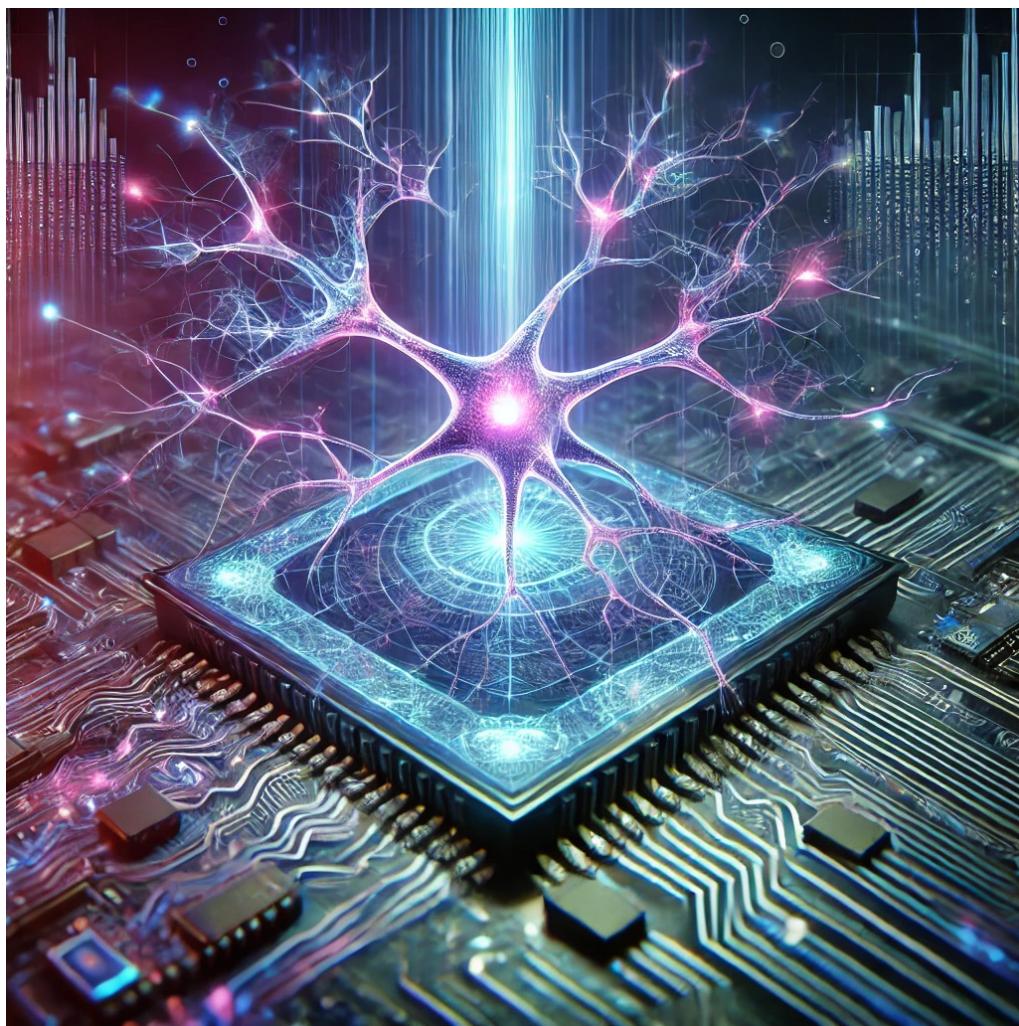


UNIDAD 4

ALGORITMOS PARA SISTEMAS DE APRENDIZAJE AUTOMÁTICO

Programación de Inteligencia Artificial
Curso de Especialización en Inteligencia Artificial y Big Data



CHATGPT prompt: Genera una imagen futurista de una neurona dentro de un ordenador

Carlos M. Abrisqueta Valcárcel
IES Ingeniero de la Cierva 2024/25

ÍNDICE

1. INTRODUCCIÓN.....	3
2. IRIS – UN DATAFRAME MUY POPULAR	4
3. CONCEPTO DE NEURONA ARTIFICIAL	7
3.1. LA PRIMERA NEURONA ARTIFICIAL: NEURONA McCULLOCH-PITTS (MCP)	8
4. EL PERCEPTRÓN	8
4.1. REGLA DE APRENDIZAJE DEL PERCEPTRÓN	10
4.2. EJEMPLO DE APRENDIZAJE	11
4.3. IMPLEMENTACIÓN EN PYTHON DE UN PERCEPTRÓN	12
4.4. LA INTERPRETACIÓN GEOMÉTRICA DEL PROCESO DE APRENDIZAJE	14
4.5. MARK I – UN EJEMPLO REAL	15
5. ADALINE – LA EVOLUCIÓN DEL SOFTWARE QUE APRENDE	15
5.1. LA OPTIMIZACIÓN DEL PROCESO DE APRENDIZAJE	17
5.2. LA TASA O RATIO DE APRENDIZAJE η	19
5.3. EL ALGORITMO DE ADALINE.....	20

1. INTRODUCCIÓN

En este tema vamos a estudiar cómo han evolucionado los algoritmos para utilizar otras técnicas basadas en estadística y matemáticas para entrenar a ordenadores a realizar problemas complejos como la regresión, clasificación o agrupamientos (clustering).

Los algoritmos de inteligencia artificial que estudiamos en el tema 3 y sus estructuras de datos asociadas, son una gran parte de lo que se ha estudiado tradicionalmente en la especialidad de inteligencia artificial, dentro del mundo de la computación. Hoy en día está de moda el machine learning, en especial, el uso de las redes neuronales artificiales para lo que se denomina aprendizaje profundo o **deep learning**.

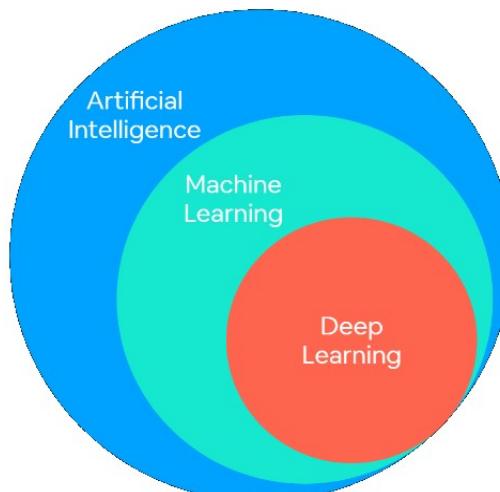


Figura 1: El campo de la IA y el encaje del machine learning en ella

En este tema, estudiaremos los algoritmos que aprenden generando una configuración en una estructura de datos en memoria que permite resolver problemas típicos de machine learning. Mientras que los algoritmos que hemos visto hasta ahora siguen un enfoque computacional tradicional, es decir, algoritmos tipo receta, donde se indican los pasos a seguir para solucionar un problema, el enfoque de este nuevo conjunto de algoritmos es completamente diferente. En lugar de proporcionar un conjunto de pasos para resolver un problema, se proporciona al algoritmo un conjunto de datos de entrada y el conjunto de datos de salida para esas entradas. El algoritmo intenta configurar una estructura de datos para generar un conjunto de reglas que, dado ese conjunto de datos de entrada, las reglas produzcan esos datos de salida esperados, o al menos, una buena aproximación de ellos. Este proceso de configurar la estructura de datos es lo que se denomina **aprendizaje**. Y la búsqueda de la configuración apropiada se llama **entrenamiento**.



Figura 2: Enfoque tradicional vs Machine Learning

En la siguiente sección, vamos a ver un ejemplo de cómo aplicar estas técnicas a través de los dos primeros algoritmos que surgieron allá por los años 50, para entender este nuevo enfoque algorítmico llamado **Machine Learning**: El **perceptrón** y su evolución natural **Adaline**.

2. IRIS – UN DATAFRAME MUY POPULAR

El conjunto de datos de **Iris**, también conocido como el conjunto de datos de **Fisher Iris**, es un conjunto de datos multivariante introducido por el estadístico y biólogo Ronald A. Fisher en 1936. Es ampliamente utilizado en el campo de la estadística y el aprendizaje automático para fines educativos y de prueba. Fisher desarrolló este conjunto de datos como un ejemplo de análisis discriminante.

El conjunto de datos de Iris contiene 150 instancias de medidas de iris. Estas medidas incluyen el largo y el ancho del sépalo y del pétalo de tres especies de iris (Iris setosa, Iris virginica e Iris versicolor), 50 muestras de cada una. Estas características se utilizan para clasificar las especies de las plantas.

El conjunto de datos de Iris consiste en varias columnas que representan las medidas y la especie de cada flor de iris. Las columnas son las siguientes:

- **Longitud del Sépalo (cm)**: Esta columna indica la longitud del sépalo de cada flor de iris, medida en centímetros.
- **Ancho del Sépalo (cm)**: Esta columna refleja el ancho del sépalo de la flor, igualmente en centímetros.
- **Longitud del Pétalo (cm)**: Esta columna mide la longitud del pétalo de la flor de iris, en centímetros.
- **Ancho del Pétalo (cm)**: En esta columna se encuentra el ancho del pétalo de la flor, medido en centímetros.
- **Especie**: Esta columna categoriza cada flor en una de las tres especies de iris: Iris setosa, Iris virginica, o Iris versicolor.

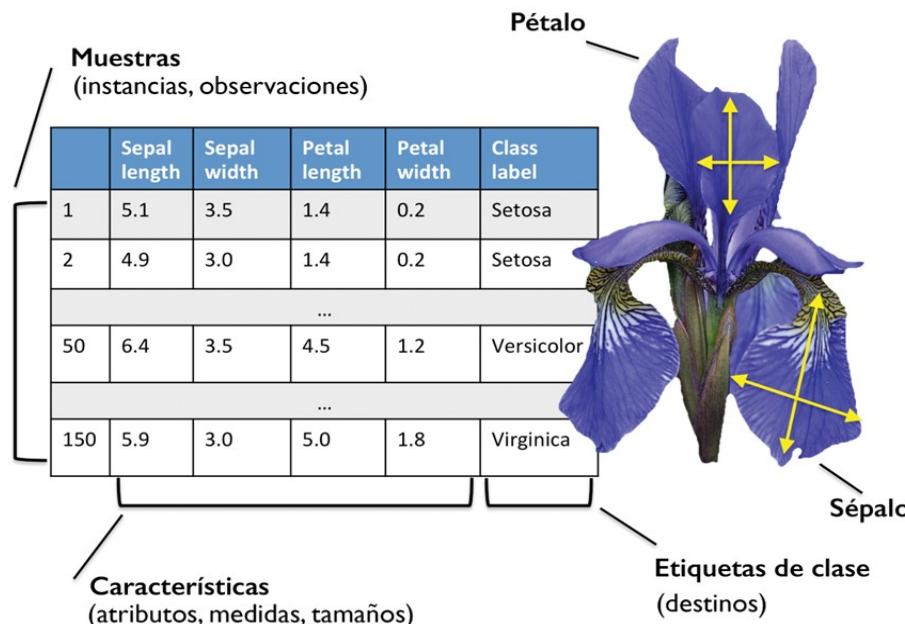


Figura 3: Conjunto de muestras para el Dataset Iris

Cada fila del conjunto de datos representa una flor de iris individual, y las mediciones de las columnas 1 a 4 son utilizadas para clasificar la flor en una de las especies mencionadas en la columna 5 (class label).

Iris se ha convertido en un caso de estudio estándar en clasificación y análisis de datos. Es particularmente útil para:

- Probar algoritmos de clasificación.
- Visualizar técnicas de reducción de dimensiones.
- Practicar análisis exploratorio de datos.

Para cargar el conjunto de datos de Iris en Python, se puede usar la función `read_csv` de la biblioteca pandas. A continuación se muestra un ejemplo de cómo hacerlo:

```
import pandas as pd
url="https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
df=pd.read_csv(url)
df
```

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

Figura 4: Carga sobre un dataframe de Iris

Para el ejemplo de clasificación que vamos a mostrar a continuación, trataremos exclusivamente dos clases, la de la variedad iris-setosa y la iris-versicolor.

```
# seleccionar setosa y versicolor (las que están en las posiciones 0:100)
y = df.iloc[0:100, 4].values
y
#salida: array de 100 posiciones:
#50 flores de tipo iris-setosa y 50 iris-versicolor
#array(['Iris-setosa', ..., 'Iris-setosa',
#       'Iris-versicolor', ..., 'Iris-versicolor'])
```

A continuación, con *numpy*, mapeamos en un array las muestras con valores -1 para setosa y 1 para versicolor. Nótese el nombre de la variable *y*, que suele denotar la clase de la observación.

```
import numpy as np
y = np.where(y == 'Iris-setosa', -1, 1)
y
#salida
#array([-1, -1, -1, -1, -1, -1, -1, -1, ..., 1, 1, 1, 1])
```

Nos quedamos con la longitud de sépalo y de pétalo: campos 0 y 2. En este caso, la variable *X* suele denotar las observaciones.

```
X= df.iloc[0:100, [0,2]].values
X
#salida
#array([[5.1, 1.4],
#       [4.9, 1.4],
#       ...]),
```

Dibujando en un gráfico de *matplotlib* podemos observar las diferencias entre las dos clases:

```
import matplotlib.pyplot as plt
# Dibujar los datos
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='versicolor')

plt.xlabel('longitud de sépalo [cm]')
plt.ylabel('longitud de pétalo length [cm]')
plt.legend(loc='upper left')

plt.savefig('clasificacion.png', dpi=300)
plt.show()
```

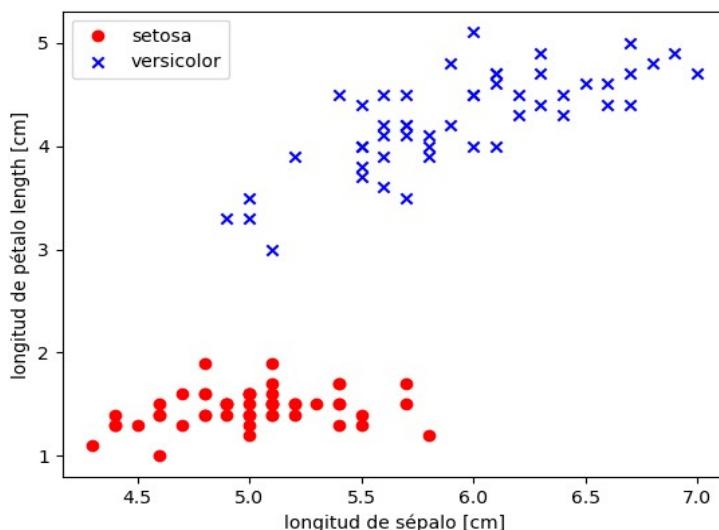


Figura 5: Gráfico matplotlib con la clasificación de Iris

Obsérvese que las dos clases son fácilmente separables por una línea recta: estamos hablando de un caso de clasificación donde las clases son linealmente separables. Cuando las muestras son linealmente separables, y de dos dimensiones (como es el caso), se puede **entrenar** un algoritmo llamado **perceptrón** para saber diferenciar las clases.

No todas las clases son linealmente separables. En la figura que se muestra a continuación, se puede observar cómo no todas las clases son linealmente separables (aunque sí separables por otros métodos):

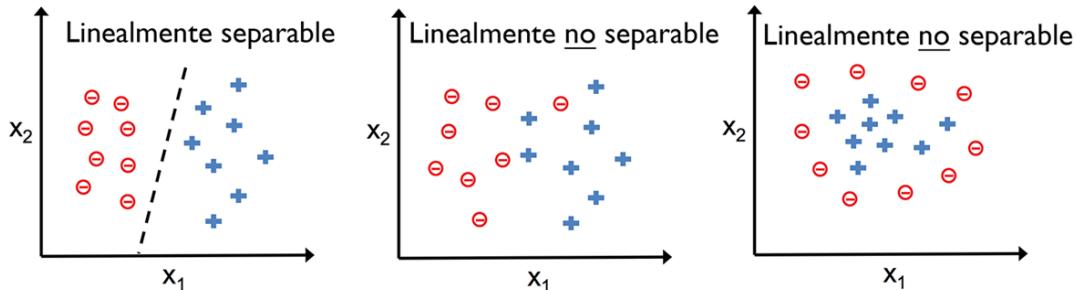


Figura 6: Clases separables y no separables linealmente

3. CONCEPTO DE NEURONA ARTIFICIAL

Una neurona es una célula especializada del sistema nervioso, cuya función principal es recibir, procesar y transmitir información a través de señales eléctricas y químicas. Estas células son fundamentales para el funcionamiento del cerebro y la medula espinal, formando redes complejas que permiten realizar tareas cognitivas y motoras. La estructura de una neurona incluye el cuerpo celular, las dendritas que reciben señales de otras neuronas, y el axón, que transmite señales a otras neuronas o tejidos.

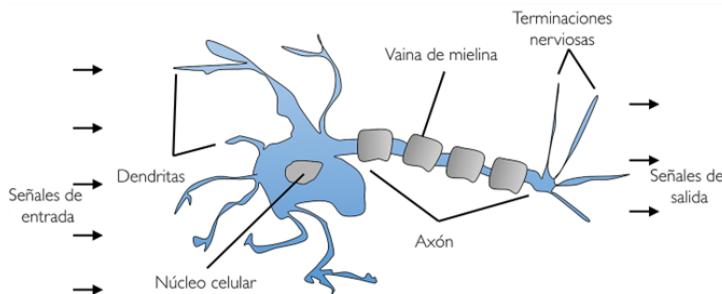


Figura 7: Neurona artificial

Santiago Ramón y Cajal, un científico español nacido en 1852, es considerado uno de los padres de la neurociencia moderna. Cajal realizó contribuciones fundamentales al entendimiento de la estructura y función del sistema nervioso. Utilizando técnicas de tinción celular, descubrió que las neuronas son entidades discretas y no una red continua, como se creía anteriormente. Esta observación, conocida como *la doctrina de la neurona*, revolucionó el entendimiento del sistema nervioso y le valió el Premio Nobel de Fisiología o Medicina en 1906, compartido con Camillo Golgi.

El trabajo de Cajal y los avances en neurociencia han tenido un impacto significativo en el desarrollo de la inteligencia artificial (IA). La comprensión de cómo las neuronas procesan y transmiten información ha inspirado el diseño de redes neuronales artificiales, que son sistemas de IA que intentan emular la forma en que el cerebro humano procesa la información. Estas redes, compuestas por unidades de procesamiento análogas a las neuronas biológicas, aprenden a realizar tareas complejas a través de ejemplos y retroalimentación, una metodología inspirada en los principios del aprendizaje y la plasticidad neuronal. El campo de la IA ha evolucionado

considerablemente desde sus inicios, y actualmente, las redes neuronales artificiales se utilizan en una amplia gama de aplicaciones, desde el reconocimiento de patrones hasta la toma de decisiones autónomas, demostrando la influencia duradera de la neurociencia en el desarrollo tecnológico.



Figura 8: Santiago Ramón y Cajal

3.1. LA PRIMERA NEURONA ARTIFICIAL: NEURONA McCulloch-Pitts (MCP)

La neurona **McCulloch-Pitts** (MCP) es un modelo simplificado de una neurona biológica, propuesto por Warren McCulloch y Walter Pitts en 1943. Esta neurona artificial es considerada uno de los primeros modelos que facilitó el desarrollo de redes neuronales. En el modelo MCP, una neurona se conceptualiza como una unidad de procesamiento binaria, que realiza una suma ponderada de sus entradas y luego aplica una función de umbral para determinar su salida.

En términos matemáticos, la salida y de una neurona MCP se define como sigue:

$$Y = f \left(\sum_{i=1}^n w_i \cdot x_i - \theta \right)$$

donde:

- x_i representa la i-ésima entrada a la neurona.
- w_i es el peso asociado con la i-ésima entrada.
- θ es el umbral de activación de la neurona.
- f es una función de activación, típicamente una función escalón, tal que:

$$f(u) = \begin{cases} 1 & \text{si } u \geq 0 \\ 0 & \text{si } u < 0 \end{cases}$$

La función de activación determina si la neurona se activa o no, en función de si la suma ponderada de las entradas menos el umbral es positiva o negativa. Este modelo, aunque simplificado en comparación con la complejidad de las neuronas biológicas, fue crucial en el establecimiento de la teoría de redes neuronales y ha influenciado el desarrollo de modelos más avanzados en el campo de la inteligencia artificial.

4. EL PERCEPTRÓN

El **perceptrón** es un algoritmo fundamental en el campo de las redes neuronales artificiales, propuesto inicialmente por Frank Rosenblatt en 1957. Se inspira en la biología de las neuronas del cerebro humano y sirve como un modelo básico para el aprendizaje supervisado en máquinas. El perceptrón consiste en una sola neurona artificial con pesos ajustables. Su funcionamiento es sencillo: recibe múltiples entradas, las cuales son multiplicadas por sus respectivos pesos, y luego sumadas.

Esta suma ponderada pasa a través de una función de activación, generalmente una función escalón, que determina la salida del perceptrón.

El aprendizaje en un perceptrón ocurre mediante la actualización iterativa de los pesos, basándose en la diferencia entre la salida deseada y la salida producida por el modelo. Este proceso se conoce como regla de aprendizaje de Rosenblatt, y es un ejemplo temprano de lo que ahora se conoce como aprendizaje supervisado en inteligencia artificial.

El modelo del perceptrón fue un avance significativo en su época y sentó las bases para el desarrollo posterior de redes neuronales más complejas. Sin embargo, en 1969, Marvin Minsky y Seymour Papert publicaron un libro titulado “Perceptrons”, donde demostraron las limitaciones del perceptrón, en particular su incapacidad para resolver problemas no linealmente separables, como la función XOR. Esto llevó a un declive en la investigación de las redes neuronales, conocido como el “invierno de la inteligencia artificial”, hasta que se desarrollaron modelos más avanzados en las décadas siguientes.

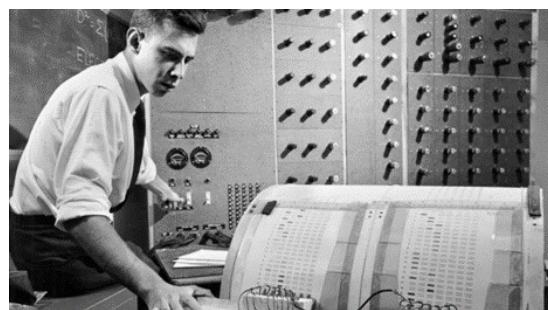


Figura 9: Frank Rosenblat (1953) y su perceptrón

A continuación mostramos cómo el perceptrón se puede utilizar para crear un clasificador: A partir de unas entradas x y unos pesos w se calcula la entrada a la red a partir de una función, se genera una salida binaria (clasificación positiva o negativa). El esquema siguiente muestra los diferentes componentes del perceptrón:

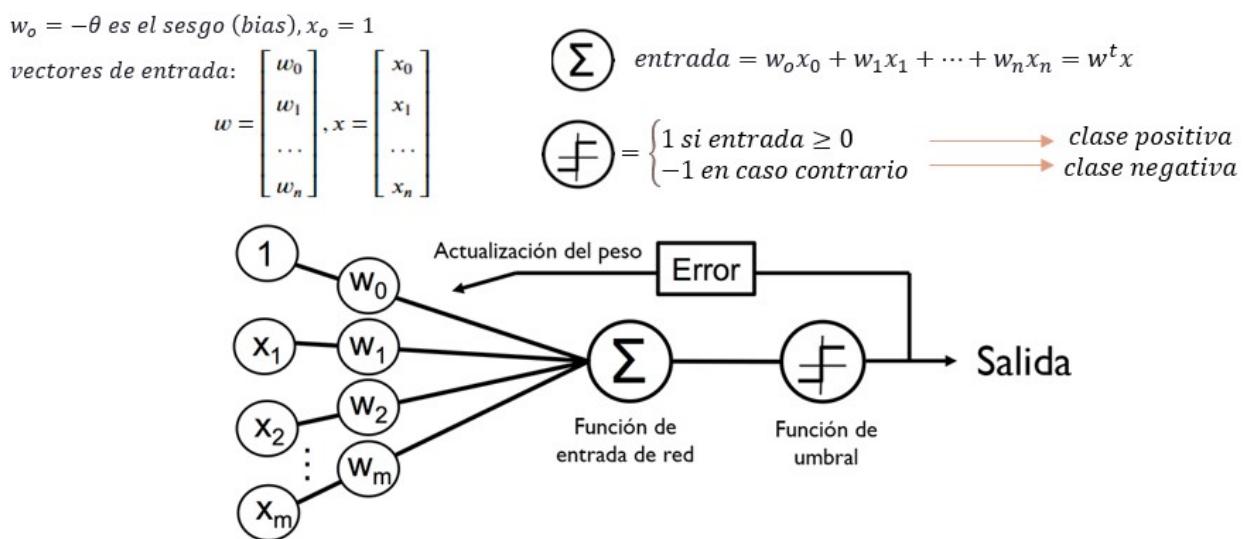


Figura 10: Esquema del perceptrón

La función de entrada a la red se calcula con la siguiente fórmula:

$$\sum_i w_i \cdot x_i = w^T \cdot x$$

La función umbral o escalón unitario, comúnmente utilizada en perceptrones, se define matemáticamente de la siguiente manera:

- La función devuelve 1 si el argumento es mayor o igual a cero.
- La función devuelve 0 si el argumento es menor que cero.

Esta función se puede definir como:

$$f(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}$$

El **sesgo** en un perceptrón es un parámetro adicional utilizado para ajustar la salida del modelo junto con los pesos de las entradas. Matemáticamente, el sesgo se puede considerar como un peso asociado a una entrada constante de valor 1. El sesgo permite que el modelo haga ajustes más finos para lograr la clasificación deseada, incluso cuando todas las entradas son cero.

Por tanto, la función de un perceptrón, incluyendo el sesgo, se define como:

$$f(x) = \phi\left(\sum_i w_i \cdot x_i + b\right)$$

donde:

- w_i son los pesos,
- x_i son las entradas,
- b es el sesgo,
- ϕ es la función de umbral o escalón unitario

La función ϕ decidirá si la entrada a la red representa una clase positiva o una clase negativa:

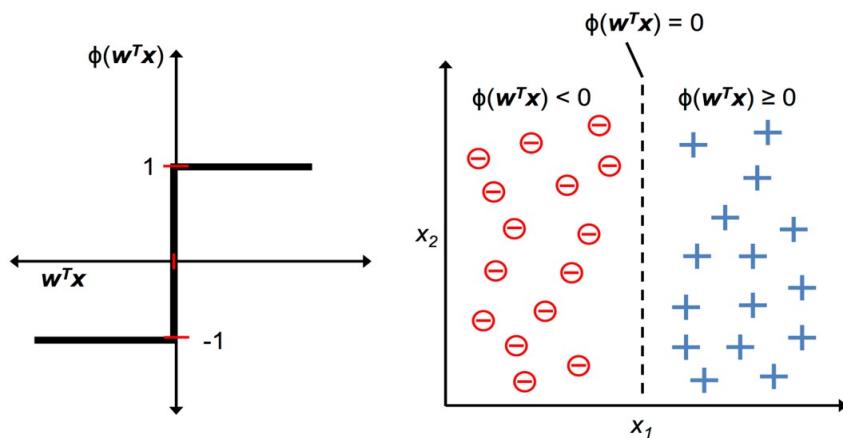


Figura 11: ϕ es la función de umbral o escalón unitario

4.1. REGLA DE APRENDIZAJE DEL PERCEPTRÓN

El algoritmo del perceptrón, también llamada *regla del aprendizaje del perceptrón de Rosenblat* es la siguiente:

- Inicializar los pesos a números aleatorios pequeños (distribución normal)

- Para cada muestra de entrenamiento (x_i)
 - Calcular el valor de salida (etiqueta predicha) y
 - Actualizar los pesos

Para actualizar los pesos, se aplica la siguiente fórmula:

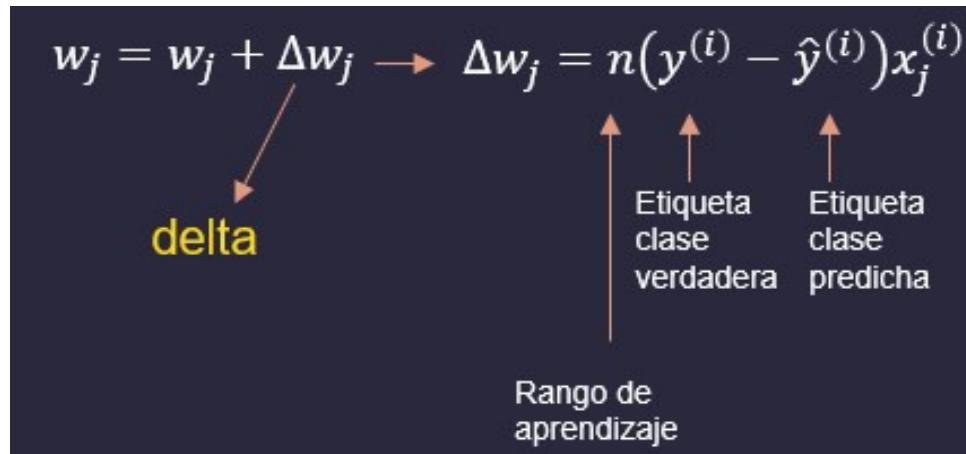


Figura 12: Actualización de los pesos del perceptrón

En la figura anterior, los parámetros clave son los siguientes:

- w_j : Representa el peso asociado a la entrada j .
- Δw_j : Es la actualización que se aplica a w_j . Se calcula como:

$$\Delta w_j = \eta \times (Etiqueta\ Verdadera^{(i)} - Etiqueta^{(i)}) \times x_j^{(i)}$$

- η : Es el factor de aprendizaje, un parámetro que determina el tamaño del paso en la actualización del peso.
- Etiqueta verdadera: El valor real de la salida que el modelo debería predecir en la iteración i .
- Etiqueta predicha: La salida que el modelo predice antes de la actualización en la iteración i .
- $x_j^{(i)}$: La entrada asociada al peso w_j del i -ésimo entrenamiento.

La regla de actualización de los pesos se realiza de acuerdo con la siguiente fórmula:

$$w_j = w_j + \Delta w_j$$

Esta regla permite que el perceptrón ajuste sus pesos para mejorar la precisión de sus predicciones en futuras iteraciones.

4.2. EJEMPLO DE APRENDIZAJE

Supongamos que el perceptrón predice correctamente una etiqueta de clase (-1): En este caso, la etiqueta verdadera será -1 y la etiqueta predicha será -1, por tanto, al aplicar la fórmula, el resultado es cero:

$$\Delta w_j = \eta(-1 - (-1)) \cdot x_j^{(i)} = 0$$

Supongamos esta vez que el perceptrón predice correctamente una etiqueta de clase (1): En este caso, la etiqueta verdadera será 1 y la etiqueta predicha será 1, por tanto, al aplicar la fórmula, el resultado es cero:

$$\Delta w_j = \eta(1 - 1) \cdot x_j^{(i)} = 0$$

Al ser $\Delta w_j = 0$, los pesos no se actualizan, es decir, el perceptrón no corrige nada, puesto que ha acertado.

Supongamos ahora que el perceptrón predice incorrectamente una etiqueta de clase (-1): En este caso, la etiqueta verdadera será 1 y la etiqueta predicha será -1, por tanto, al aplicar la fórmula, el resultado es distinto de cero:

$$\Delta w_j = \eta(-1 - 1) \cdot x_j^{(i)} = \eta \cdot (-2 x_j^{(i)})$$

Por tanto, al actualizar el peso, también se empuja al perceptrón hacia la clase correcta en la siguiente iteración.

4.3. IMPLEMENTACIÓN EN PYTHON DE UN PERCEPTRÓN

El siguiente algoritmo implementa un perceptrón que se utiliza para entrenar un clasificador binario capaz de distinguir entre las flores de tipo iris setosa e iris versicolor del dataframe popular descrito en el punto 2:

```
class Perceptron(object):
    """
    Parámetros
    -----
        ratio : float, ratio de aprendizaje (entre 0.0 y 1.0)
        n_iter: int, Pasadas sobre el dataset de entrenamiento. (también se llaman épocas)
        semilla : int, Semilla para el generador de números aleatorios para inicializar el vector de pesos de forma aleatoria

    Atributos
    -----
        w: 1d-array, Pesos después del entrenamiento.
        errors: list, Número de clasificaciones erróneas (updates) en cada época
    """

    def __init__(self, ratio=0.01, n_iter=50, semilla=1):
        self.ratio = ratio
        self.n_iter = n_iter
        self.semilla = semilla
        rgen = np.random.RandomState(semilla)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.errors_ = []

    def fit(self, X, y):
        """ entrenar el conjunto de datos.
```

```

Parámetros
-----
X: {array-like}, shape = [n_muestras, n_caracteristicas]
    Vectores de entrenamiento, donde n_muestras es el número de muestras
y n_caracteristicas es el número de características.
y: array-like, shape = [n_muestras] Valores objetivo (target).

Devuelve
-----
self : object
"""

for _ in range(self.n_iter):
    errors = 0
    for xi, target in zip(X, y):
        update = self.ratio * (target - self.predict(xi))
        self.w_[1:] += update * xi
        self.w_[0] += update
        errors += int(update != 0.0)
    self.errors_.append(errors)
return self

def net_input(self, X):
    """Calcula la entrada a la red"""
    return np.dot(X, self.w_[1:]) + self.w_[0]
#+ self.w_[0]*1 en realidad

def predict(self, X):
    """Retorna la etiqueta de clase después de un paso"""
    return np.where(self.net_input(X) >= 0.0, 1, -1)

#entrenar el perceptrón
p=Perceptron(0.1,10,1)
p.fit(X,y)

plt.plot(range(1, len(p.errors_)+1 ), p.errors_, marker='o')
plt.xlabel('épocas')
plt.ylabel('número de actualizaciones')
plt.show()

#predecir una en concreto
sepalo=3
petalo=5
print("la flor con longitud de sépalo",sepalo,"y longitud de pétalo", petalo,
      "es",np.where(p.predict([sepalo, petalo])==1,'versicolor','setosa'))

```

La clase contiene parámetros y métodos para el entrenamiento del perceptrón y la realización de predicciones.

Parámetros

- **ratio:** Tasa de aprendizaje, un valor flotante entre 0.0 y 1.0
- **n_iter:** Número de pasadas sobre el conjunto de datos de entrenamiento, también conocidas como épocas (epochs)
- **semilla:** Semilla para el generador de números aleatorios, usada para la inicialización aleatoria de los pesos.

Atributos

- **w_:** Vector de pesos después del entrenamiento.
- **errors_:** Lista del número de clasificaciones erróneas en cada época.

Métodos

- **fit:** Método para entrenar el perceptrón con un conjunto de datos.
- **net_input:** Calcula la entrada a la red neuronal.
- **predict:** Realiza la predicción de la etiqueta de clase.

4.4. LA INTERPRETACIÓN GEOMÉTRICA DEL PROCESO DE APRENDIZAJE

La interpretación geométrica del perceptrón, especialmente en el contexto de la clasificación de dos tipos de flores iris (setosa y versicolor), es bastante interesante.

Estamos usando el perceptrón para la clasificación binaria. La idea básica es que este algoritmo intenta dibujar una línea (o hiperplano en dimensiones más altas) que separa las dos clases de datos. En el caso del conjunto de datos de iris, estamos considerando dos clases: setosa y versicolor.

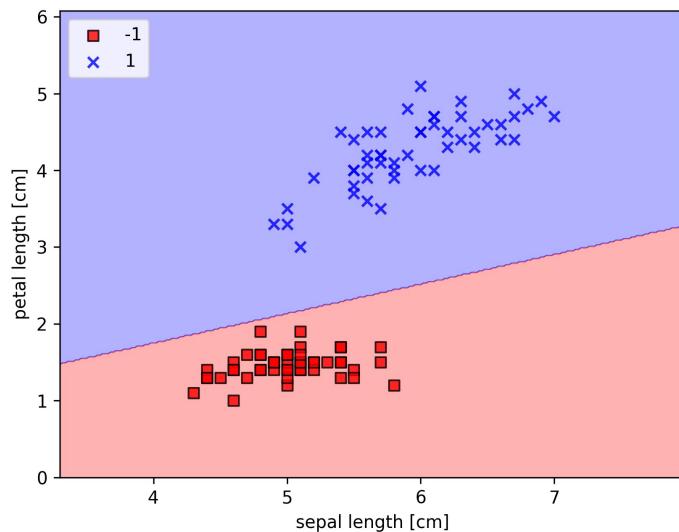


Figura 13: Separación de las regiones de decisión

Imagina que tienes un gráfico en 2D donde cada eje representa una característica de las flores (por ejemplo, el largo y el ancho del sépalo). Cada punto en este gráfico representa una flor, y el objetivo es separar las setosas de las versicolor.

La “línea de separación” creada por el perceptrón es una línea en este espacio 2D que intenta separar los puntos correspondientes a setosa de los que corresponden a versicolor. Los puntos a un lado de la línea se clasifican como una clase (por ejemplo, setosa), mientras que los puntos al otro lado se clasifican como la otra clase (versicolor).

La posición y la orientación de esta línea se determinan durante el proceso de entrenamiento del perceptrón. El algoritmo ajusta los parámetros de la línea (su pendiente y su intersección con el eje) basándose en los datos de entrenamiento para minimizar los errores de clasificación.

4.5. MARK I – UN EJEMPLO REAL

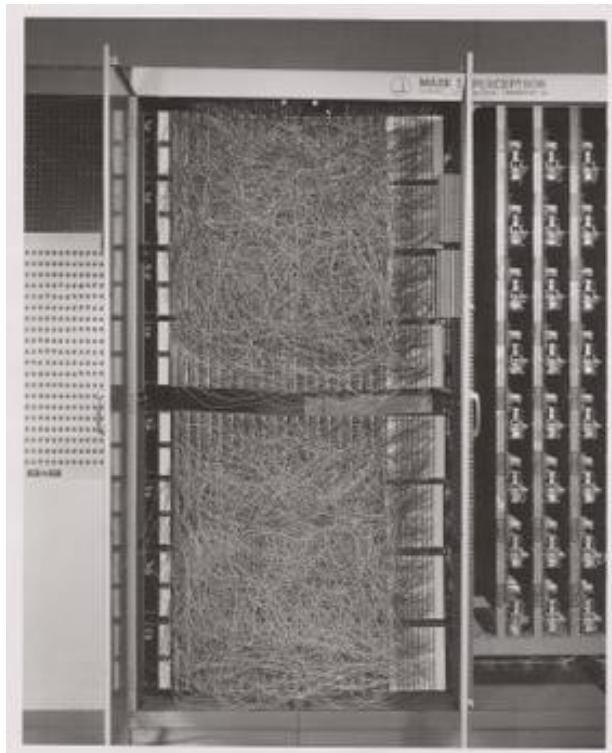


Figura 14: Mark I perceptron

La *Mark I Perceptron* del *Cornell Aeronautical Laboratory*, creada por *Frank Rosenblatt* en la década de 1950, se diseñó con el propósito de explorar las posibilidades de las máquinas para simular el procesamiento de información biológica, especialmente la capacidad de reconocer patrones y aprender de la experiencia. La máquina fue una de las primeras implementaciones de una red neuronal artificial y fue construida para demostrar los principios del aprendizaje automático, con la esperanza de que pudiera eventualmente adquirir la capacidad de realizar cualquier tarea intelectual que un ser humano pueda hacer. Se utilizó principalmente en tareas de clasificación visual, demostrando que las máquinas podrían ajustar sus parámetros internos (pesos sinápticos) en respuesta a estímulos del entorno y mejorar su rendimiento en tareas cognitivas con el tiempo.

5. ADALINE – LA EVOLUCIÓN DEL SOFTWARE QUE APRENDE

A pesar de los avances que el perceptrón de Rosenblatt representó para la computación y el aprendizaje automático, sufría de limitaciones significativas, especialmente su incapacidad para resolver problemas que no son linealmente separables, como el problema del XOR. Esta limitación fue resaltada por Marvin Minsky y Seymour Papert en su libro “Perceptrons”, lo que llevó a un

decrecimiento en el interés por las redes neuronales durante un tiempo. En respuesta a estas limitaciones, se desarrolló el modelo **ADALINE** (ADAptive LInear NEuron) por Bernard Widrow y Ted Hoff. ADALINE fue diseñado como un modelo de neurona lineal con una regla de aprendizaje basada en la minimización del error cuadrático, lo cual representaba un avance respecto al perceptrón al proporcionar una solución algorítmica para el ajuste de pesos que se basaba en una función de coste diferenciable, la cual permitía el uso de métodos de optimización más eficientes.

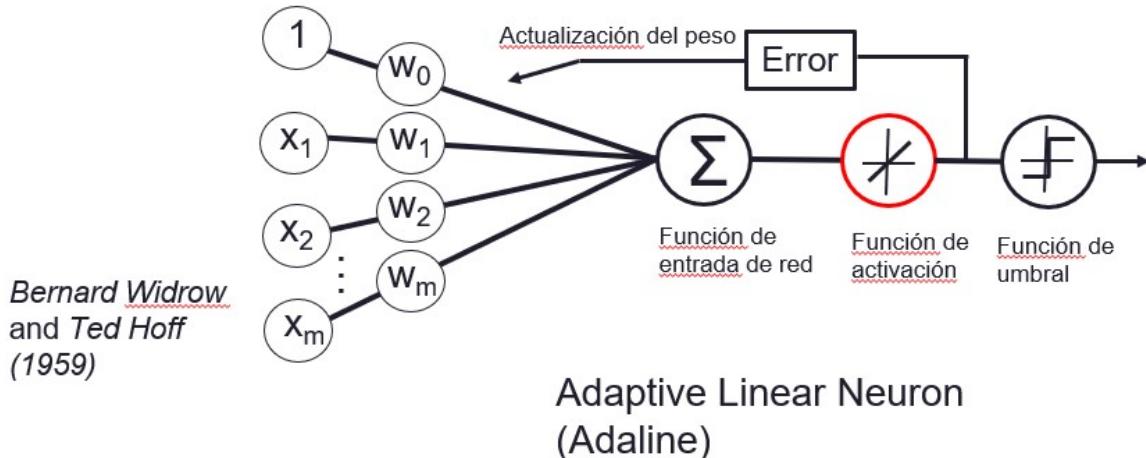


Figura 15: El esquema de ADALIN (ADAptive LInear NEuron)

Uno de los conceptos novedosos que introduce Adaline es el de función de activación. Esta función permite el aprendizaje basado en la minimización de una función de coste continua y diferenciable. Esto posibilita la optimización del proceso de aprendizaje, por ejemplo, con el uso de algoritmos como el del descenso del gradiente para ajustar los pesos, lo que es esencial para la convergencia del algoritmo en problemas linealmente separables.

Al utilizar una función de activación lineal, ADALINE puede calcular la salida real que es una suma ponderada de las entradas y luego aplicar una función de umbral para tomar una decisión de clasificación final. Sin embargo, la actualización de los pesos se realiza utilizando la salida lineal, lo que facilita el cálculo de los gradientes y la aplicación de cambios incrementales a los pesos en la dirección que más reduce el error cuadrático medio. Esto hace que ADALINE sea particularmente bueno para problemas de regresión y clasificación binaria donde la linealidad es una suposición razonable.

Las diferencias entre el perceptrón y Adaline podemos resumirlas en los siguientes puntos:

- ADALINE utiliza el método de ajuste por mínimos cuadrados de las predicciones de una función de activación lineal (coste: media de los errores al cuadrado).
- El proceso de aprendizaje se realiza basado en la salida de una función de activación lineal $\phi(z)$ en lugar de la de escalón unitario.
- El perceptrón actualiza los pesos calculando la diferencia entre la clase verdadera y y la predicción \hat{y} : sólo aprende cuando falla en una predicción.
- ADALINE actualiza los pesos calculando la diferencia entre la clase esperada y y la salida de la función de activación \hat{y} , que es un valor real: Puede aprender incluso cuando no ha fallado la predicción.
- Además, matemáticamente, esto permite minimizar una función continua de coste asociado al aprendizaje.

5.1. LA OPTIMIZACIÓN DEL PROCESO DE APRENDIZAJE

La función de coste, también conocida como función de pérdida o error, es una medida de cómo de bien el modelo está realizando su tarea. ADALine utiliza una función de coste conocida como media de los errores al cuadrado o error cuadrático medio (SSE, por sus siglas en inglés o ECM, por sus siglas en castellano). Matemáticamente, se expresa como:

$$SSE = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

donde y_i es el valor real y \hat{y}_i es el valor predicho por el modelo para la i -ésima instancia.

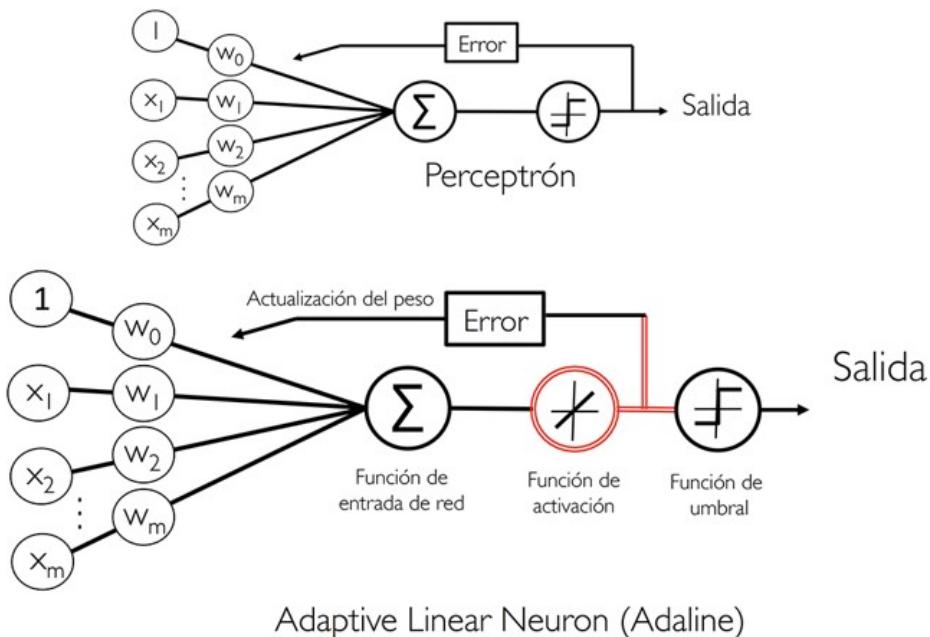


Figura 16: Diferencia entre Adaline y el Perceptrón

Características de la función de coste:

- **Diferenciable:** Esta propiedad es crucial ya que permite el uso del algoritmo de descenso del gradiente para optimizar la función.
- **Convexa:** Asegura que la solución encontrada es un mínimo global, lo que es particularmente importante en modelos lineales como ADALine.
- **Penaliza errores mayores:** Al elevar al cuadrado la diferencia, errores mayores tienen un impacto más significativo en la función de coste, lo que ayuda a mejorar la precisión del modelo.

Importancia: La función de coste es fundamental en el proceso de aprendizaje de ADALine. Permite cuantificar el error del modelo y, a través de su optimización, mejorar el rendimiento. ADALine utiliza un algoritmo llamado descenso del gradiente para su optimización, cuyo objetivo es encontrar el mínimo de la función de coste o *global mínima*.



Figura 17: El descenso del gradiente

Este algoritmo es fundamental para encontrar los pesos óptimos que minimizan la función de coste, en este caso, el Error Cuadrático Medio (ECM). Este algoritmo itera sobre los datos de entrenamiento y ajusta los pesos y el sesgo en la dirección que reduce el ECM. En cada iteración, los pesos se actualizan en la dirección opuesta al gradiente de la función de coste con respecto a esos pesos, lo cual se logra mediante la expresión, donde η es la tasa de aprendizaje. Este proceso iterativo permite que ADALINE aprenda gradualmente la relación lineal entre las características de entrada y las salidas deseadas, optimizando así el rendimiento del modelo en la tarea de clasificación o regresión.

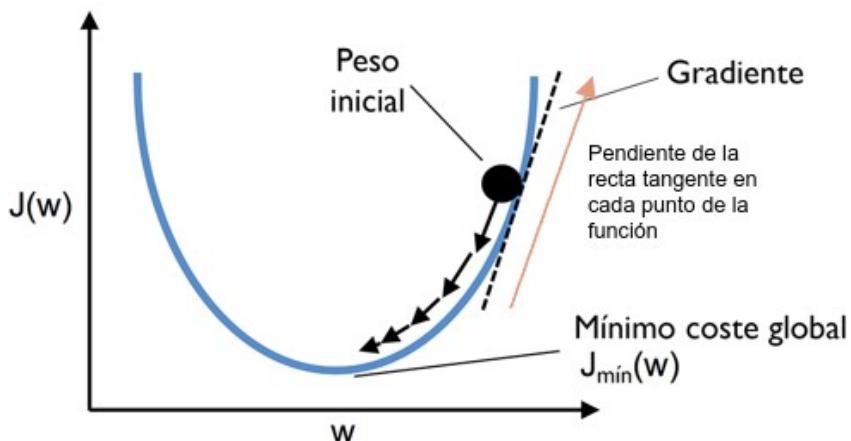


Figura 18: Búsqueda del mínimo global mediante la derivada de la función de coste

Para optimizar ADALINE, se deriva la función de coste para medir el error entre las salidas predichas y las reales en un conjunto de datos. La derivación se realiza en varios pasos.

- **Definición de la Función de Coste:** La función de coste ECM para ADALINE se define como la suma del cuadrado de las diferencias entre las salidas predichas y_{pred} y las salidas reales y_{real} , dividida entre dos veces el número de muestras (n). Matemáticamente, se expresa como:

$$ECM = \frac{1}{2n} \sum_{i=1}^n (y_{pred}^{(i)} - y_{real}^{(i)})^2$$

- **Salida Predicha:** La salida predicha en ADALINE se calcula como una combinación lineal de las entradas y los pesos, más un sesgo. Si $x^{(i)}$ es el vector de entrada para la muestra i , w es el vector de pesos, y b es el sesgo, entonces:

$$y_{pred}^{(i)} = w^T \cdot x^{(i)} + b$$

- **Derivación del ECM:** Para actualizar los pesos en ADALINE, se necesita calcular el gradiente del ECM con respecto a cada peso. Esto implica tomar la derivada parcial del ECM con respecto a cada peso w_j y el sesgo b . La derivada parcial con respecto a un peso w_j es:

$$\frac{\partial ECM}{\partial w_i} = \frac{1}{n} \sum_{i=1}^n (y_{pred}^{(i)} - y_{real}^{(i)}) \cdot x_j^{(i)}$$

y para el sesgo b :

$$\frac{\partial ECM}{\partial b} = \frac{1}{n} \sum_{i=1}^n (y_{pred}^{(i)} - y_{real}^{(i)})$$

- **Actualización de Pesos:** Los pesos se actualizan en dirección opuesta al gradiente para minimizar el ECM. Esto se hace típicamente utilizando un método como el descenso de gradiente, donde cada peso se actualiza como

$$w_j = w_j - \eta \cdot \frac{\partial ECM}{\partial w_j}$$

donde η es la tasa de aprendizaje.

Podemos observar cómo esta expresión es idéntica a la regla de aprendizaje del perceptrón, es decir, la salida de la función de activación coincide con la entrada. Esto significa que la salida de la función de activación es simplemente una transformación lineal de la entrada, sin aplicar ninguna no linealidad. En términos simples, la salida de la red es igual a la suma ponderada de sus entradas, lo que hace que ADALine sea adecuado para problemas donde la relación entre entrada y salida es aproximadamente lineal.

Puedes examinar este artículo de Sebastian Rascha donde detalla el proceso de cálculo de esta derivada de la función de coste:

<https://sebastianraschka.com/faq/docs/linear-gradient-derivative.html>

5.2. LA TASA O RATIO DE APRENDIZAJE η

La tasa de aprendizaje, denotada como η representa la velocidad a la que el algoritmo aprende. Específicamente, controla el tamaño de los pasos que se toman para alcanzar el mínimo de la función de coste.

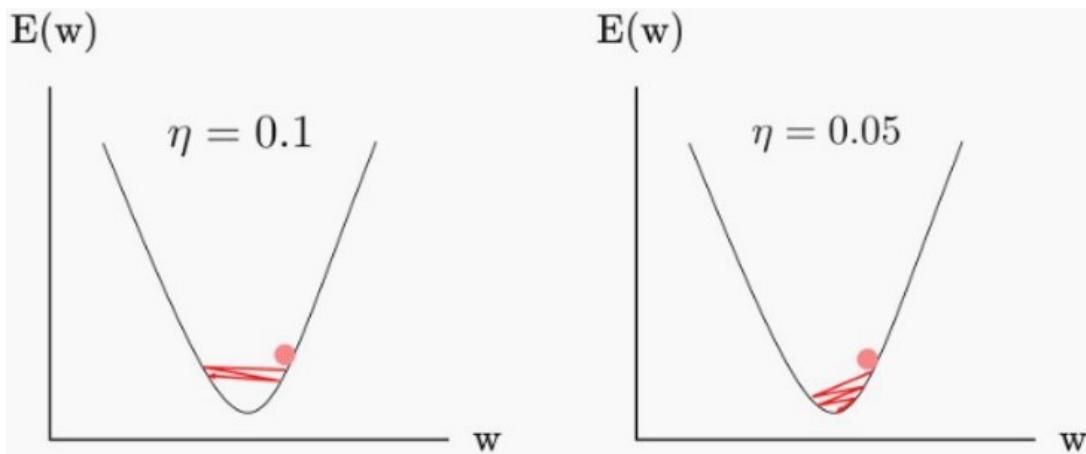


Figura 19: Tasa de aprendizaje en el descenso del gradiente

Un valor de η muy grande puede causar que el algoritmo sobrepase el mínimo provocando lo que se conoce como overshooting, mientras que un valor demasiado pequeño puede resultar en un proceso de aprendizaje muy lento o en quedar atrapado en mínimos locales, *local mínima*. Por lo tanto, elegir un valor adecuado para η es fundamental para asegurar una convergencia eficiente y efectiva durante el entrenamiento del modelo.

5.3. EL ALGORITMO DE ADALINE

El algoritmo de descenso del gradiente para ADALINE comienza con la inicialización de los pesos, incluido el sesgo como w_0 , con valores pequeños aleatorios. La tasa de aprendizaje η se establece para controlar el tamaño de los ajustes en los pesos durante el entrenamiento. En cada iteración, el algoritmo recorre todas las muestras del conjunto de entrenamiento, calculando la salida predicha, $y_{pred}^{(i)}$, como el producto del vector de pesos transpuesto w^T y el vector de entrada $x^{(i)}$. Para cada peso w_j , incluyendo el sesgo w_0 , se realiza una actualización en la dirección opuesta al gradiente de la función de coste (Error Cuadrático Medio). Este proceso se repite hasta que se cumple un criterio de parada, como un número fijo de iteraciones o una convergencia en el valor del ECM, asegurando así el aprendizaje del modelo a partir de los datos.

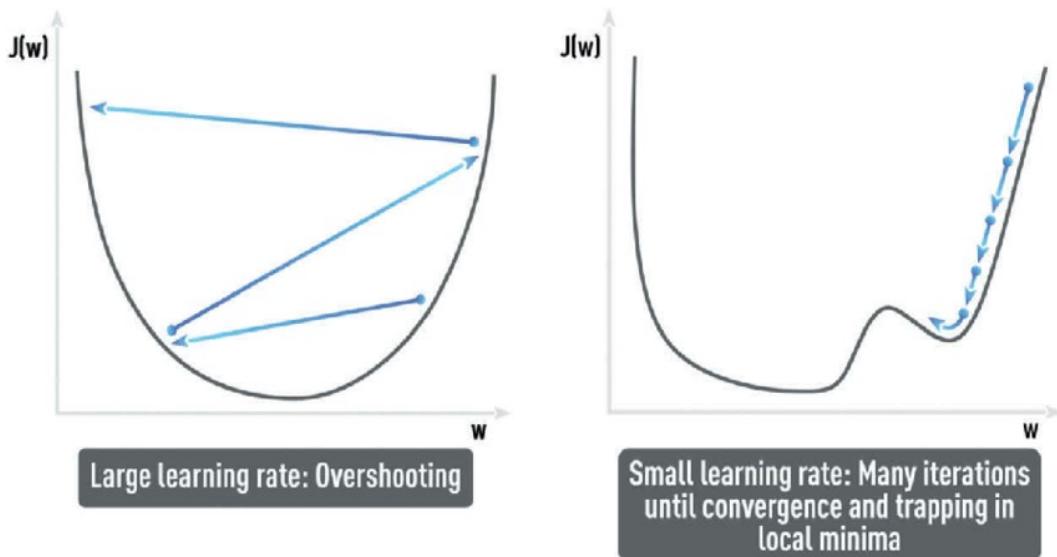


Figura 20: Overshooting vs local minima

Algoritmo 1: Algoritmo de Descenso del Gradiente para ADALINE con Sesgo como w_0

```
Iniciar los pesos w incluyendo  $w_0$  (sesgo) con valores pequeños aleatorios
Establecer la tasa de aprendizaje  $\eta$ 
while no se cumpla el criterio de parada do
    for cada muestra i en el conjunto de entrenamiento do
        Calcular la salida predicha  $y_{pred}^{(i)} = w^T x^{(i)}$  (considerando  $x_0^{(i)} = 1$  para el sesgo)
        for cada peso  $w_j$ , j = 0 hasta m do
            Actualizar  $w_j := w_j - \eta \cdot \frac{1}{n} \sum_{i=1}^n (y_{pred}^{(i)} - y_{real}^{(i)}) \cdot x_j^{(i)}$ 
        end for
    end for
    Evaluar la función de coste ECM
end while
```