

Índice

Introducción	2
Contexto	3
Tipos de Aprendizaje en Redes Neuronales Artificiales	4
¿Qué es una neurona?	4
Etapas de la estimación de recursos minerales donde es posible usar RNA	7
Estudios anteriores	9
Teoría	11
Bases de la neurona artificial	11
Neurona de McCulloch & Pitts.....	11
El Perceptrón simple	16
Implementación en python del perceptrón simple	19
Perceptrón simple monocapa para clasificar múltiples clases	24
Implementación perceptrón simple para clasificar zonificación mineral	26
Perceptrón multicapa.....	29
Arquitectura	29
Propagación	30
Función de costo	31
Función de Optimización	34
Retropropagación	35
Implementación del Perceptrón multicapa en python	40
Función de Activación en Redes Neuronales Artificiales.....	45
Funciones de optimización	49
Selección de Hiperparámetros	56
Comparación de RNAs con estimación geoestadística	60
Implementación	61
Implementación en python: definición de unidades geológicas.....	61
Implementación en python: Incorporación de ley al modelo	61
Visualización con SGeMS	61

Introducción

La estimación de recursos minerales es una de las etapas primarias de la minería, lo que permite cuantificar y caracterizar los yacimientos, y, por lo tanto, es necesaria para su explotación eficiente y sostenible. Tradicionalmente, las estimaciones de recursos minerales se han derivado utilizando técnicas geoestadísticas, tales como la interpolación espacial y el kriging, que utilizan datos obtenidos de perforaciones para modelar la distribución de concentraciones de mineral en un depósito. Aunque estos métodos están bien establecidos y han demostrado buen rendimiento, tienen algunas limitaciones, particularmente en el caso de depósitos complejos o con distribución mineral no lineal.

En este sentido, las redes neuronales artificiales (RNA) se plantean como una alternativa innovadora para la mejora de la estimación de recursos minerales. Las redes neuronales artificiales (RNA) son sistemas computacionales inspirados en las redes neuronales biológicas que constituyen el cerebro animal y, como tales, pueden aprender relaciones complejas entre variables a la manera del aprendizaje automático; por esa razón, las RNA son una de las técnicas que se están utilizando para abordar problemas que tienen relaciones altamente no lineales y heterogéneas.

Se llaman artificiales porque están inspiradas en las neuronas biológicas que son las unidades básicas de nuestro cerebro y las que reciben, procesan y transmiten la información. Una neurona biológica comprende un soma (cuerpo celular), dendritas que recogen señales, un axón que transmite impulsos eléctricos y terminales axónicos que facilitan la comunicación con otras neuronas a través de sinapsis. De manera similar a esto, las RNA tienen un conjunto de nodos (neuronas artificiales) en varias capas, y la conexión entre ellas imita el intercambio de información en el cerebro humano.

Por lo tanto, las RNA han demostrado gran potencial para la estimación de recursos minerales, con la capacidad de captar relaciones no lineales que otros métodos no logran, y pueden usar varias variables geológicas juntas sin hacer suposiciones estadísticas fuertes. Introducidas por McCulloch y Pitts en 1943, las RNA se han convertido en una de las metodologías principales de aprendizaje profundo y sus aplicaciones cubren una serie de campos, como la minería.

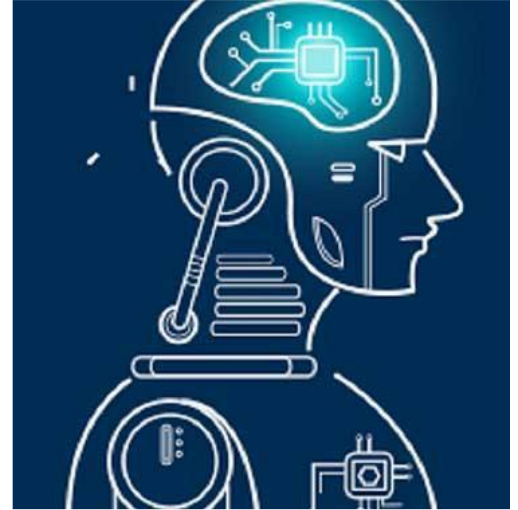
El objetivo de este trabajo es delinear más claramente dónde las redes neuronales artificiales se utilizan mejor en un flujo de trabajo de estimación de recursos minerales, comparar su rendimiento con las técnicas geoestadísticas tradicionales y determinar cuándo pueden considerarse aplicables. Se discutirá la arquitectura de varios modelos de RNA y su capacidad predictiva en función de la cantidad y calidad de los datos disponibles, mostrando el potencial de esta tecnología en la industria minera.

Contexto

En esta sección se buscará visualizar el contexto de las redes neuronales artificiales definiendo las ideas clave y mostrando aplicaciones potenciales.

1. Inteligencia Artificial (IA)

La Inteligencia Artificial es el campo de estudio que tiene como objetivo crear máquinas inteligentes que realicen tareas que requieren inteligencia humana. Está impulsada por sistemas que pueden aprender de experiencias, reconocer objetos, comprender y responder al lenguaje, tomar decisiones y resolver problemas. Su objetivo es crear sistemas que vean, razonen, actúen en el mundo y, a veces, sean creativos.



2. Aprendizaje Automático (AA o ML - Machine Learning)

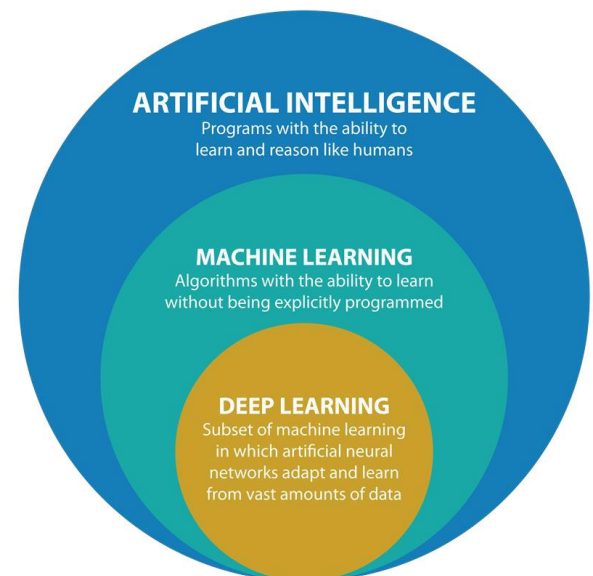
El aprendizaje automático es una subdisciplina de la IA que permite que las máquinas aprendan a partir de datos sin ser explícitamente programadas. Los modelos se interfazan con sus datos a través de algoritmos, por lo que, a medida que un modelo gana experiencia, cambia sus parámetros y mejora su rendimiento. Mientras que la IA busca imitar la inteligencia humana en general, el ML busca enseñar a las máquinas a reconocer patrones en los datos y hacer predicciones o asignaciones.

3. Aprendizaje Profundo (AP o DL - Deep Learning)

El aprendizaje profundo es un tipo específico de ML que emplea el uso de redes neuronales artificiales, consistentes en múltiples capas, para modelar datos con patrones complejos. A diferencia de organizar datos en conjuntos para ejecutarlos a través de ecuaciones predefinidas, el DL establece parámetros en capas con mínima intervención humana, lo que permite que la computadora aprenda por sí misma cuando se expone a los datos, confiando en esas capas de procesamiento para comprender los datos.

Diferencias clave con el ML:

- El ML requiere mayor intervención humana, ya que los modelos deben ajustarse manualmente.
- El DL es más autónomo, ya que la red neuronal aprende de los datos sin intervención constante.
- Mayor demanda computacional, pero mayor capacidad de detectar patrones complejos.



Tipos de Aprendizaje en Redes Neuronales Artificiales

Las Redes Neuronales Artificiales (RNA) pueden aprender de diferentes maneras según la disponibilidad de datos etiquetados. Los principales enfoques son:

Aprendizaje Supervisado:

En este caso, el modelo se entrena con un conjunto de datos previamente etiquetado, es decir, donde se conoce el valor del atributo objetivo (por ejemplo, ley de cobre). La RNA ajusta sus parámetros en función de ejemplos con entradas y salidas conocidas, permitiendo realizar tareas como clasificación y regresión.

Ejemplo:

- Predicción de la ley del mineral en zonas no perforadas a partir de datos de sondeos.
- Clasificación de litologías según datos geoquímicos y geofísicos.

Aprendizaje No Supervisado:

En este caso, dado que el modelo opera con datos no etiquetados, no sabe de antemano cuál debería ser el resultado. En otras palabras, el objetivo es identificar patrones ocultos o agrupar datos similares (agrupamiento) en lugar de predecir valores particulares. Una de sus aplicaciones es en la definición de dominios geológicos.

¿Qué es una neurona?

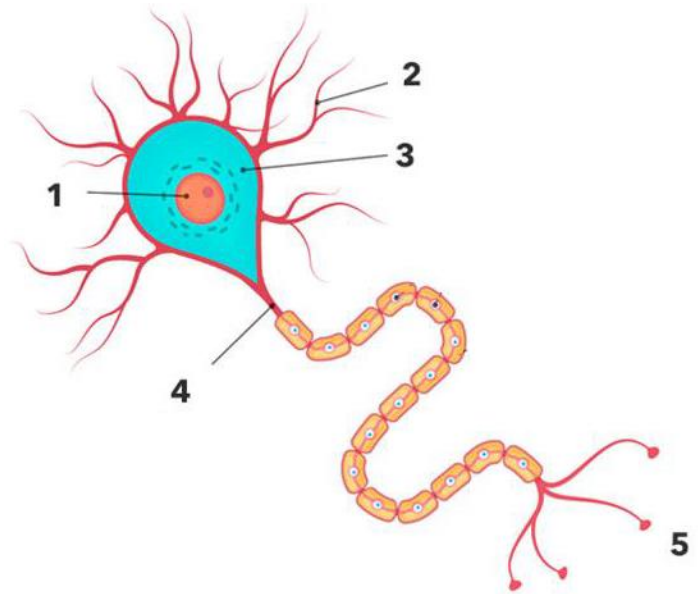
Las neuronas son las unidades principales del cerebro, encargadas de recibir, procesar y transmitir toda la información. Explícitamente, son células que forman parte del sistema nervioso y desempeñan un papel fundamental en la comunicación neuronal. Se estima que el cerebro humano contiene aproximadamente 100 mil millones de neuronas, las cuales trabajan en conjunto para procesar información y generar respuestas.

El cerebro aprende en función de las experiencias, lo que permite a los seres humanos reconocer patrones y reaccionar rápidamente ante estímulos familiares. Por ejemplo, una persona puede identificar el rostro de un familiar entre un grupo grande de personas en cuestión de segundos, debido a que su cerebro ha aprendido y almacenado esa información a lo largo del tiempo.

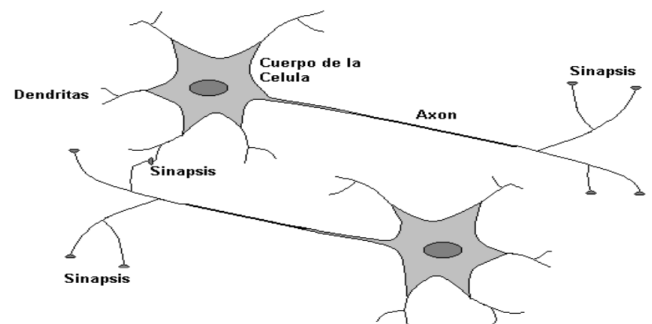
Desde hace aproximadamente 80 años, la humanidad ha intentado emular el comportamiento del cerebro a través de modelos matemáticos que permitan resolver problemas complejos. Este esfuerzo ha llevado al desarrollo de las redes neuronales artificiales, que buscan replicar el funcionamiento de las neuronas biológicas y su capacidad de aprendizaje mediante algoritmos avanzados de inteligencia artificial.

Estructura general de una neurona:

1. Núcleo: Es la parte central de la neurona, se encuentra situada en el cuerpo celular y se encarga de producir energía para el funcionamiento de la neurona.
2. Dendritas: "brazos de la neurona", forman pequeñas prolongaciones ramificadas que salen de las diferentes partes del soma de la neurona. Su función principal es la recepción de estímulos provenientes de otras neuronas
3. Soma: Cuerpo celular. Esta es la parte donde se fabrican la mayoría de las moléculas de la neurona y donde se llevan a cabo los procesos vitales de mantenimiento celular y funcionamiento de la célula nerviosa en sí.
4. Axón: Es una fina y alargada fibra nerviosa envuelta en vainas de mielina que se encarga de transmitir las señales eléctricas desde el soma de la neurona hasta los botones terminales.
5. Terminal de axón: También se conocen como botones sinápticos; están ubicados al final del axón y se dividen en terminales; su función es conectarse con otras neuronas y formar sinapsis.



Sinapsis Neuronal: La neurona está formada por una estructura cuyas partes principales son el núcleo, el cuerpo celular y las dendritas. Entre estas existen numerosas conexiones gracias a sus axones, es decir sus pequeñas ramificaciones. Los axones ayudan a crear redes cuya función es transmitir mensajes de neurona en neurona. Este proceso es denominado como sinapsis.

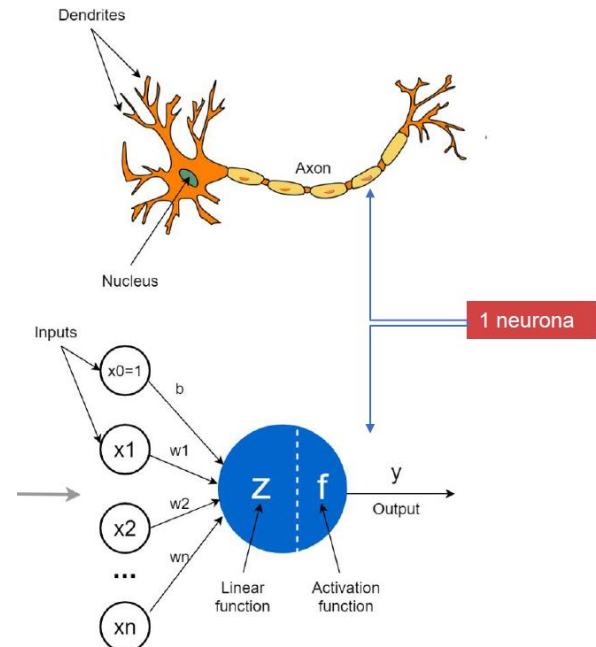


Redes neuronales artificiales (RNA)

Las redes neuronales artificiales son modelos computacionales que emulan el funcionamiento del cerebro humano, procesando información mediante un conjunto de nodos interconectados, conocidos como neuronas artificiales. Estas fueron introducidas en 1943 por McCulloch, neurólogo, y Pitts, lógico.

Su estructura se compone de capas de neuronas que transmiten señales desde la entrada hasta generar una salida, ajustando sus conexiones a través del entrenamiento. Utilizan funciones de activación para modelar relaciones complejas y pueden manejar datos no lineales de manera eficiente, pero eso se explicará a detalle más adelante.

Cada neurona artificial funciona de manera similar a una neurona biológica. Las entradas (x) representan los datos que la red recibe y pueden compararse con las dendritas, que captan señales en una neurona biológica. Luego, estas señales se combinan y procesan dentro de la neurona, donde se realiza una sumatoria ponderada, similar a cómo el núcleo de una neurona procesa la información antes de generar una respuesta. Posteriormente, la función de activación determina si la señal se transmite o no, de forma análoga al axón en la neurona biológica, que solo envía impulsos si se alcanza un cierto umbral. Finalmente, la salida de la neurona artificial se comunica a las siguientes capas, como los terminales del axón transmiten señales a otras neuronas o músculos en el cuerpo humano.

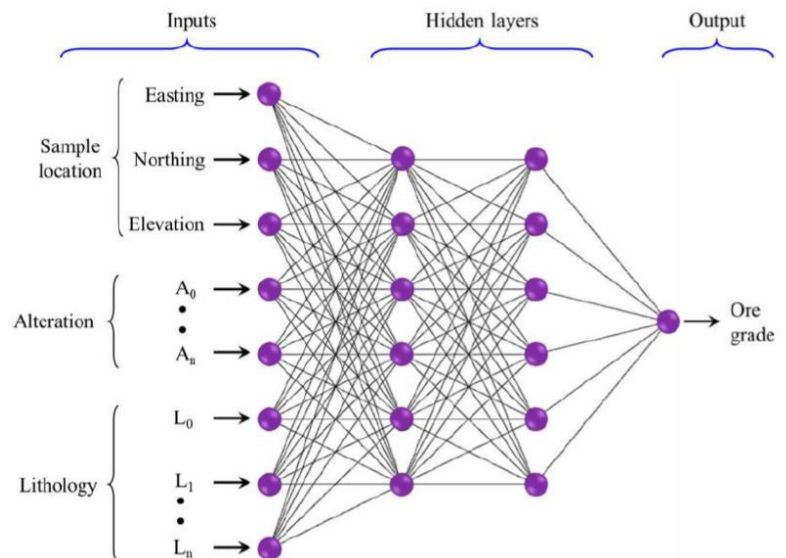


Arquitectura de una red neuronal artificial

La arquitectura de una red neuronal artificial (RNA) define cómo se organizan sus neuronas y conexiones. En general, está compuesta por tres tipos de capas:

1. **Capa de entrada (Input layer):** Recibe los datos del problema a resolver. Por ejemplo, se pueden ingresar datos geológicos, como ubicación, tipo de roca, contenido mineral, etc. Cada variable es representada por un nodo en esta capa.
2. **Capas ocultas (Hidden layers):** Son las responsables del procesamiento de la información.

Cada neurona en estas capas aplica una transformación matemática (mediante pesos,



sesgos y funciones de activación) para extraer patrones y relaciones en los datos. La cantidad de capas ocultas y neuronas define la complejidad del modelo. A partir de 2 capas ocultas ya se considera aprendizaje profundo.

3. **Capa de salida (Output layer):** Genera el resultado final del modelo, por ejemplo, la ley de un mineral en función de los datos de entrada.

El flujo de información en la red ocurre a través de conexiones ponderadas entre neuronas, ajustadas durante el entrenamiento mediante algoritmos de aprendizaje, como la retro propagación. Esto permite que la RNA aprenda a realizar predicciones precisas con base en ejemplos previos.

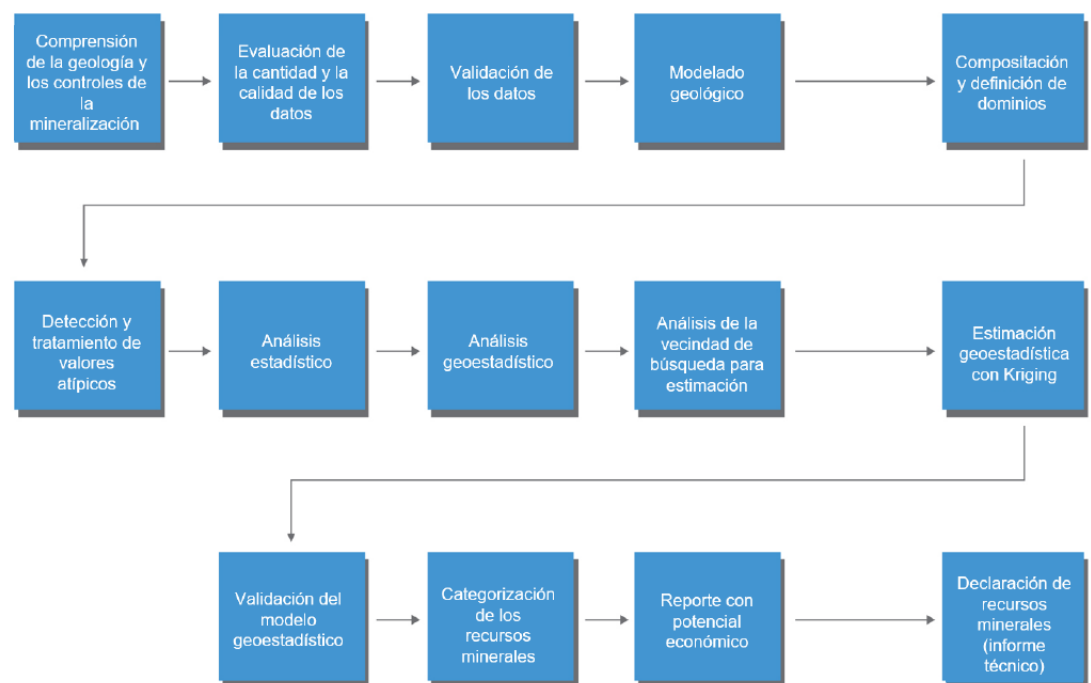
Etapas de la estimación de recursos minerales donde es posible usar RNA

En la imagen se observa el proceso tradicional de estimación de recursos minerales utilizando geoestadística clásica. Este enfoque se basa en la teoría de los variogramas y la interpolación espacial para predecir la distribución de la mineralización en un yacimiento.

El proceso comienza con la comprensión geológica y la identificación de los controles de mineralización, lo que permite definir el contexto en el que se desarrollará el análisis. Luego, se procede a la evaluación de la cantidad y calidad de los datos,

asegurando que la información utilizada sea representativa y confiable. Posteriormente, los datos son validados para detectar posibles errores o inconsistencias antes de utilizarlos en el modelado.

Una vez validados, se realiza el modelado geológico, donde se definen las características estructurales del depósito mineral y se establecen los dominios geológicos, que segmentan la mineralización en unidades homogéneas. Después de esta etapa, se identifican y tratan los valores atípicos, evitando que sesguen la estimación.

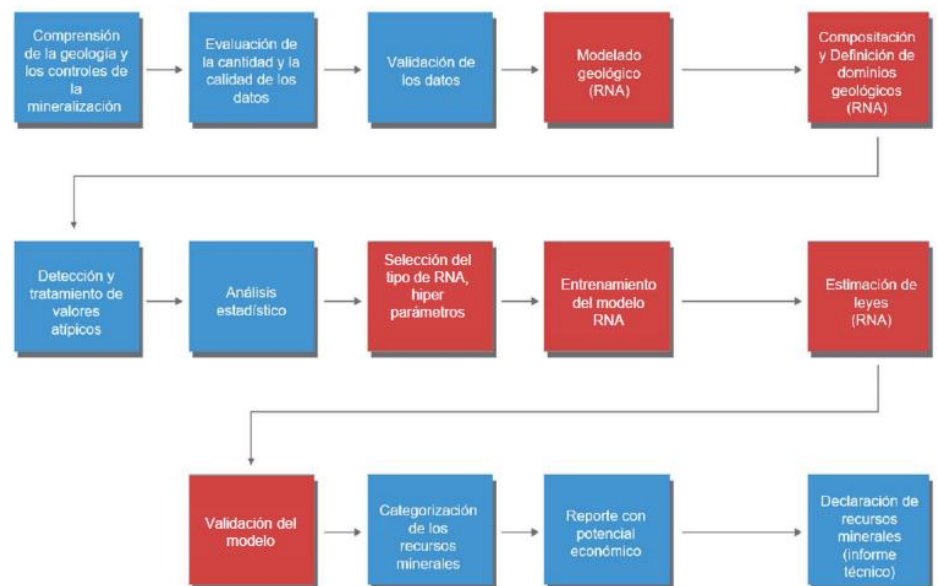


A continuación, se llevan a cabo análisis estadísticos y geoestadísticos para caracterizar la distribución espacial del recurso. Esto incluye el análisis de la vecindad de búsqueda y la aplicación de técnicas de interpolación como el kriging, que permite estimar los valores en zonas no muestreadas basándose en la correlación espacial de los datos.

El modelo resultante es validado para evaluar su precisión y confiabilidad. Finalmente, se realiza la categorización de los recursos minerales en medidos, indicados e inferidos, considerando la incertidumbre y la confiabilidad de la estimación, seguido de un reporte con el potencial económico del recurso. Como último paso, se emite la declaración de recursos minerales en un informe técnico oficial.

En la segunda imagen, el flujo de trabajo se modifica con la introducción de Redes Neuronales Artificiales (RNA) para la estimación de recursos minerales. En este caso, aunque las primeras etapas del proceso (comprensión geológica, evaluación y validación de datos) se mantienen, los métodos tradicionales de interpolación son reemplazados por modelos de aprendizaje automático.

La mayor diferencia radica en que, en lugar de definir explícitamente un modelo geoestadístico, se selecciona un tipo de RNA y se ajustan sus hiperparámetros para que el modelo aprenda a partir de los datos disponibles. Luego, el modelo se entrena con los datos históricos para identificar patrones y relaciones complejas en la distribución espacial del recurso.



Este enfoque tiene el potencial de adaptarse mejor a estructuras geológicas complejas o no lineales, aunque requiere una cuidadosa validación del modelo para garantizar su confiabilidad. Finalmente, se procede a la categorización de los recursos y a la elaboración del informe técnico, como en el enfoque tradicional.

Estudios anteriores

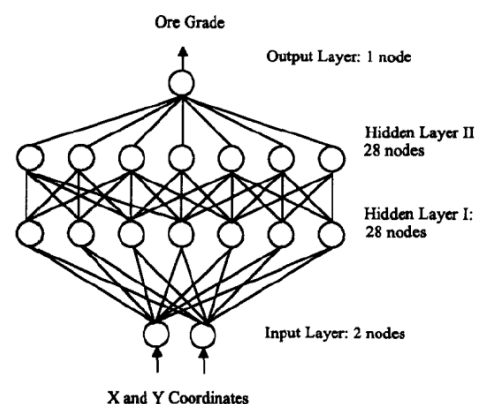
A continuación, se presentan diversos trabajos relevantes que exploran la aplicación de redes neuronales artificiales (RNA) y técnicas de aprendizaje automático en la estimación de recursos minerales:

1. Reserve Estimation Using Neural Network Techniques

Autores: Wu & Zhou (1993)

Resumen:

Este estudio propone una RNA del tipo perceptrón multicapa *feedforward* para estimar la ley de cobre (Cu%) a partir de coordenadas espaciales bidimensionales (X, Y). El modelo incluye dos capas ocultas, lo que permite considerarlo como una arquitectura de *deep learning*. Las variables de entrada son normalizadas en el rango [0, 1] antes de su ingreso al modelo. Se utilizan 51 sondajes como base de datos para el entrenamiento.

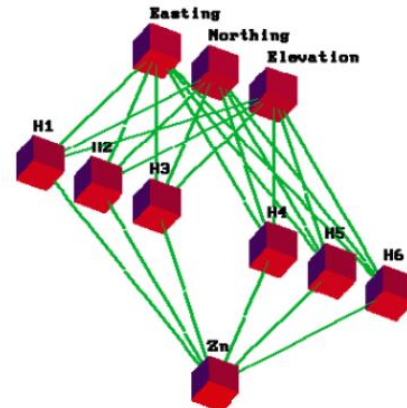


2. Application of Artificial Neural Network Systems to Grade Estimation from Exploration Data

Autor: Kapageridis (1999)

Resumen:

Tesis doctoral presentada en la University of Nottingham. Incluye un capítulo dedicado a la aplicación de RNA en minería. El autor argumenta que, si bien la geoestadística ha sido ampliamente adoptada durante las últimas décadas como herramienta principal para la evaluación de yacimientos, esta requiere numerosos supuestos, conocimientos especializados y tiempo para ser implementada eficazmente. Las redes neuronales se presentan como una alternativa más automatizada, flexible y menos restrictiva en cuanto a supuestos teóricos.

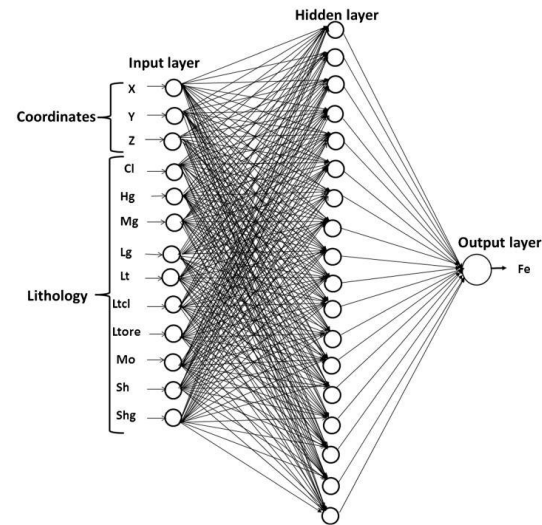


3. **Investigation of General Regression Neural Network Architecture for Grade Estimation of an Indian Iron Ore Deposit**

Autores: Goswami et al. (2017)

Resumen:

El estudio analiza la estimación de leyes de hierro utilizando coordenadas espaciales tridimensionales (X, Y, Z) y unidades litológicas múltiples como variables de entrada. Se emplean tanto perceptrones multicapa del tipo *feedforward* como Redes Neuronales de Regresión Generalizada (GRNN), una variante de las redes de base radial propuesta por D.F. Specht (1991).



4. **Application of Machine Learning to Resource Modelling of a Marble Quarry with DomainMCF**

Autor: Kapageridis (2021)

Resumen:

Este trabajo presenta el uso de *DomainMCF*, una herramienta desarrollada por la empresa Maptek que aplica tecnología basada en redes neuronales artificiales para el modelado de recursos en una cantera de mármol. Se destaca la automatización del proceso de modelado geológico mediante inteligencia artificial.



5. **Estimation of Mineral Resources with Machine Learning Techniques**

Autores: Galetakis et al. (2022)

Resumen:

Los autores estiman leyes de cobre utilizando una red neuronal de propagación hacia adelante (*forward propagation*), con coordenadas espaciales tridimensionales (X, Y, Z) como entradas del modelo. Los resultados son comparados con los obtenidos mediante kriging, a través de curvas de tonelaje, con el fin de evaluar la efectividad de los métodos de aprendizaje automático.

6. **Machine Learning Based Systems Application to Mineral Resource Estimation and Compliance with Reporting Codes for Mineral Resources**

Autores: Kapageridis et al. (2021)

Resumen:

Se analiza la aplicabilidad de algoritmos automáticos, incluyendo RNA, en la estimación de recursos minerales. El enfoque se centra en cómo estos métodos cumplen con los requisitos de los códigos de reporte internacionalmente aceptados, tales como JORC y NI 43-101, destacando el avance de los sistemas basados en *machine learning* hacia la estandarización industrial.

Teoría

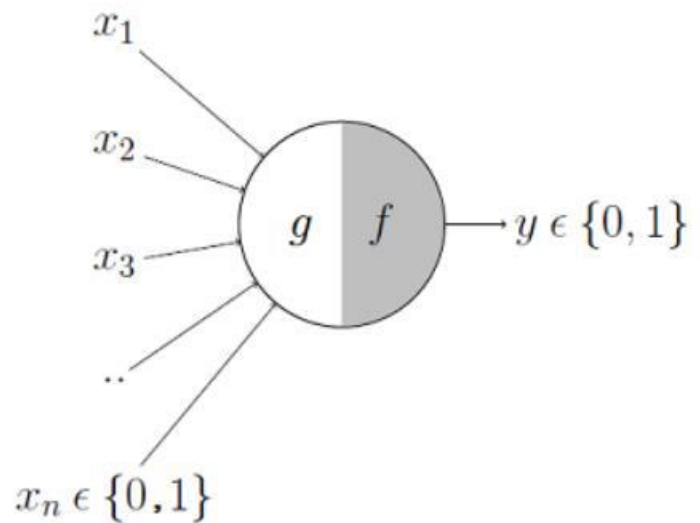
Bases de la neurona artificial

Neurona de McCulloch & Pitts

El primer modelo computacional de una neurona fue propuesto en 1943 por Warren McCulloch (neurocientífico) y Walter Pitts (lógico).

Este modelo está compuesto por una capa de entrada con un conjunto de variables x_n de tipo binario. Estas variables ingresan a la neurona, que se divide en dos partes:

1. **Suma de entradas (g):** Se calcula la sumatoria de las variables de entrada.
2. **Función de activación (f):** Se aplica una función escalón con un umbral θ que determina la activación de la neurona. Si la suma de las entradas supera θ , la neurona se activa y emite una salida binaria.



$$g(x_1, x_2, x_3, \dots, x_n) = g(\mathbf{x}) = \sum_{i=1}^n x_i$$

Las entradas pueden ser:

- **Excitatorias:** Contribuyen a la activación de la neurona.
- **Inhibitorias:** Si una entrada inhibitoria está presente, la neurona no se activará, sin importar el resto de las entradas.

$$y = f(g(\mathbf{x})) = \begin{cases} 1 & \text{if } g(\mathbf{x}) \geq \theta \\ 0 & \text{if } g(\mathbf{x}) < \theta \end{cases}$$

Características:

- Solo tiene dos estados: apagado (0) y encendido (1).
- No cuenta con un término de sesgo (bias).

- Se entrena mediante prueba y error.
- Puede resolver casi cualquier problema aritmético y lógico.
- No puede resolver problemas linealmente no separables, como la compuerta XOR.

Dado que las entradas y salidas son binarias, este modelo es útil para representar operaciones lógicas simples, pero tiene limitaciones en problemas más complejos.

Ejemplo 1:


En este ejemplo, se requiere que la muestra contenga cobre y provenga del cuerpo mineral para ser considerada como mineral de interés. Se puede representar mediante valores binarios, donde 1 indica una respuesta afirmativa y 0 una negativa.

La clasificación se realiza en dos pasos:

1. Cálculo de la suma de entradas (g).
2. Aplicación de un umbral (θ): Si la suma g es mayor o igual a θ , el resultado es 1 (mineral de interés); de lo contrario, es 0.

x1: ¿la muestra tiene cobre?

x2: ¿la muestra proviene dentro del cuerpo mineral modelado?

x1	x2	mineral		g(x)	y
0	0	0		0	0
0	1	0		1	0
1	0	0		1	0
1	1	1		2	1

A continuación, se presentan distintos valores de umbral y sus efectos en la clasificación:

g	f	umbral
0	1	0
1	1	
1	1	
2	1	

Aquí, todos los valores son clasificados como mineral de interés, lo que puede generar falsos positivos.

g	f	umbral
0	0	1
1	1	
1	1	
2	1	

Con un umbral de 1, la clasificación es más precisa, descartando muestras sin contenido de cobre ni origen mineral.

g	f	umbral
0	0	1.1
1	0	
1	0	
2	1	

g	f	umbral
0	0	2
1	0	
1	0	
2	1	

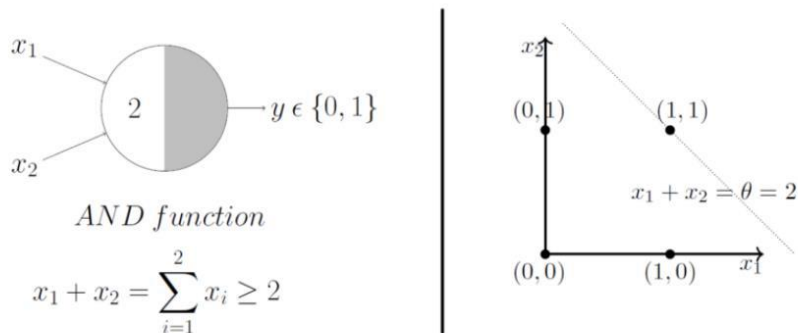
En estos 2 casos, el umbral se ha ajustado correctamente, clasificando como mineral de interés solo a la última muestra, que cumple ambas condiciones.

g	f	umbral
0	0	2.1
1	0	
1	0	
2	0	

Si el umbral es demasiado alto, todas las muestras son descartadas, lo que genera falsos negativos.

Si consideramos un umbral de 2, la ecuación que separa las muestras de interés de las no interesantes es: $x_1 + x_2 \geq 2$

Esto representa la función lógica AND, donde solo se acepta una muestra si ambas condiciones son verdaderas.



Ejemplo 2:

En este caso, el criterio de clasificación es más flexible: una muestra será considerada mineral de interés si al menos una de las condiciones es positiva (es decir, si contiene cobre o proviene del cuerpo mineral).

Siguiendo el mismo procedimiento que en el ejemplo anterior:

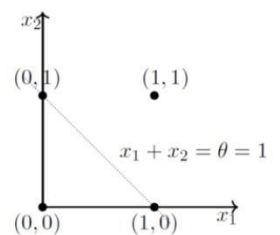
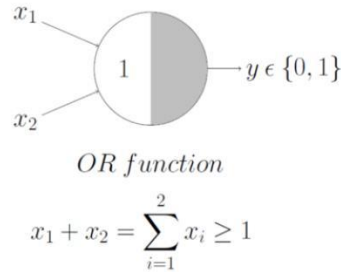
x1: ¿la muestra tiene cobre?
x2: ¿la muestra proviene dentro del cuerpo mineral modelado?

x1	x2	mineral	g(x)	y
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	2	1



Aquí, con un umbral de 1, cualquier muestra que tenga al menos una condición positiva es aceptada, lo que representa la función lógica OR.

Si consideramos un umbral de 1, la ecuación que separa las muestras de interés de las no interesantes es: $x_1 + x_2 \geq 1$

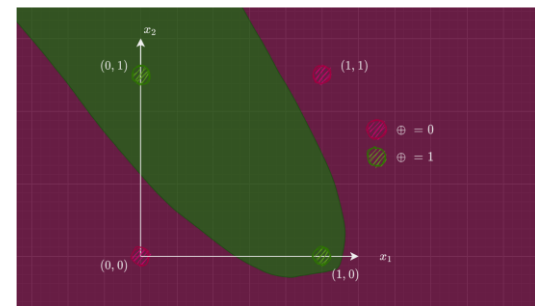


Esto significa que cualquier muestra que cumpla al menos una de las condiciones será clasificada como mineral de interés.

Estos ejemplos, aunque básicos, son fundamentales para empezar a comprender el entrenamiento en una Red Neuronal Artificial (RNA). Permiten visualizar cómo se toman decisiones a partir de datos y etiquetas objetivo, sentando las bases para problemas más complejos en clasificación y reconocimiento de patrones.

Limitaciones:

- Solo aceptan valores binarios (0 y 1), lo que restringe su aplicación a problemas que puedan representarse de esta manera.
- El umbral debe ajustarse manualmente, lo que puede dificultar la optimización del modelo.
- Todas las entradas tienen el mismo peso, impidiendo asignar mayor importancia a ciertos valores.
- No pueden resolver problemas que no sean linealmente separables, como la compuerta XOR de la imagen, donde es afirmativo solo si uno de los valores es positivo.



A continuación, se implementa en python, teniendo en cuenta que como no está creada en ninguna librería por su poca funcionalidad se debe crear de cero, en verde esta explicada cada línea:

```
#Se importan las librerías necesarias
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score

class MPNeuron:

    def __init__(self): # Inicializa el umbral de activación como None (aún no se ha definido)
        self.threshold = None

    def model(self, x): # Esta es la función que aplica el perceptrón, sumando los inputs y comparando con el umbral
        z = sum(x) # Calcula la suma de los valores de entrada
        return (z >= self.threshold) # Si la suma es mayor o igual al umbral, devuelve True, sino False

    def predict(self, X): # Predice la salida para un conjunto de entradas
        Y = []
        for x in X: # Para cada conjunto de entradas, se aplica el modelo y se guarda el resultado
            result = self.model(x)
            Y.append(result) # Agrega la predicción al conjunto de resultados
        return np.array(Y) # Devuelve las predicciones en formato de arreglo de NumPy

    def fit(self, X, Y): # Ajusta el umbral del perceptrón para maximizar la precisión
        accuracy = {} # Diccionario para almacenar la precisión para cada valor de umbral
        for th in range(X.shape[1] + 1): # Recorre posibles valores de umbral de 0 a número de características
            self.threshold = th # Establece el umbral en el valor actual
            Y_pred = self.predict(X) # Predice las salidas para los datos de entrada X
            accuracy[th] = accuracy_score(Y_pred, Y) # Calcula la precisión y la guarda
        # La función fit no retorna nada, pero tiene la información de la precisión por umbral en el diccionario 'accuracy'
```

```
mp_neuron = MPNeuron() # Crea un objeto de la clase MPNeuron, que representa el perceptrón

# Creamos un DataFrame con las entradas de la neurona (conjunto de datos de entrenamiento)
df_int = pd.DataFrame({"x1": [0, 0, 1, 1], "x2": [0, 1, 0, 1]})
# Creamos un DataFrame con las salidas deseadas para las entradas anteriores
df_out = pd.DataFrame({"salida": [0, 1, 1, 1]})

df1 = df_int.to_numpy() # Convierte el DataFrame de entradas (df_int) en un arreglo de NumPy
df2 = df_out.to_numpy() # Convierte el DataFrame de salidas (df_out) en un arreglo de NumPy

# Ajustamos el modelo utilizando el método 'fit' y los datos de entrada (df1) y salida (df2)
mp_neuron.fit(df1, df2)

# Mostramos el umbral que la neurona ha encontrado para maximizar la precisión
mp_neuron.threshold |
```

✓ 0.0s

2

Vemos que como resultado se obtiene el umbral de 2 previamente encontrado. Modificando x1 y x2 se obtienen los otros ejemplos.

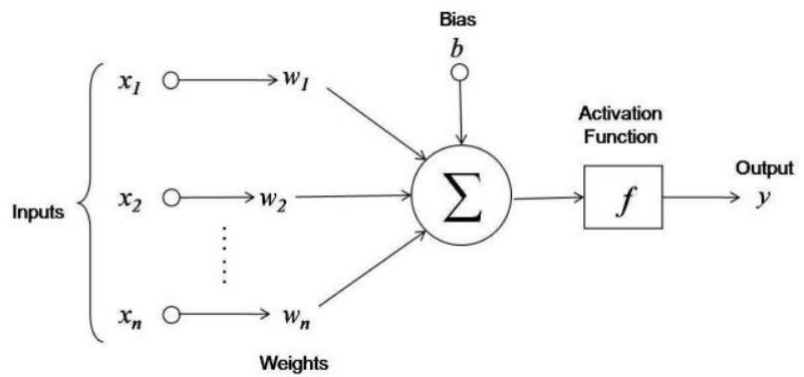
El Perceptrón simple

El Perceptrón es un modelo de neurona artificial propuesto por el psicólogo Frank Rosenblatt en 1958, basado en el modelo de McCulloch y Pitts. Su principal innovación fue la introducción de una regla de aprendizaje basada en la corrección del error, lo que permitía que la neurona ajustara sus pesos sinápticos automáticamente a partir de ejemplos de entrenamiento.

En 1969, Marvin Minsky y Seymour Papert realizaron un análisis detallado del Perceptrón, demostrando sus capacidades y limitaciones.

Características del Perceptrón Simple

- Es un modelo de neurona artificial que mejora la propuesta de McCulloch y Pitts al permitir entradas numéricas, tanto continuas como discretas.
- Se basa en una unidad lógica de umbral (*Threshold Logic Unit* o TLU), la cual computa una suma ponderada de las entradas y la compara con un umbral.
- Utiliza una función de activación escalón (*Heaviside*), que devuelve 1 si la suma ponderada es mayor o igual al umbral y 0 en caso contrario.
- Se emplea en problemas de clasificación binaria.



Funcionamiento del Perceptrón Simple

El Perceptrón Simple consta de una única capa de unidades TLU. Cada neurona realiza el siguiente cálculo:

$$z(x) = \sum_{i=1}^n x_i w_i + b$$

Donde:

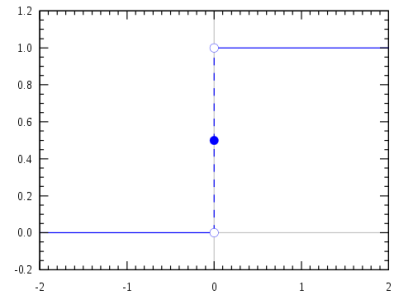
- x_i son las entradas de la neurona.
- w_i son los pesos sinápticos.

- b es el sesgo (*bias*), que permite ajustar la función de activación.
- $z(x)$ es la suma ponderada de las entradas.

La función de activación escalón de Heaviside se define como:

$$f(z) = \begin{cases} 0, & \text{si } z \leq 0 \\ 1, & \text{si } z > 0 \end{cases}$$

Esto significa que la neurona solo se activará si la suma ponderada supera un cierto umbral.



Entrenamiento del Perceptrón

El entrenamiento se basa en la regla de actualización de pesos:

$$bc = (\eta (y - y^*))$$

$$\Delta wi = \eta (y - y^*) xi$$

Donde:

- η es la tasa de aprendizaje.
- y es la salida esperada.
- y^* es la salida calculada por el perceptrón.
- Δwi es la corrección aplicada a cada peso.
- bc es la corrección del sesgo.
- X_i valor de entrada conocido

El Δwi y bc se suman a los valores iniciales, obteniendo un nuevo valor de pesos y bias a probar. El proceso se repite hasta que el modelo logra clasificar correctamente todas las instancias de entrenamiento.

Ejemplo

Volvemos a resolver el ejemplo anterior en el que, si se cumple x_1 o x_2 se clasifica como mineral, pero en este caso usando el perceptrón.

x1: ¿la muestra tiene cobre?

x2: ¿la muestra proviene dentro del cuerpo mineral modelado?

x1	x2	mineral
0	0	0
0	1	1
1	0	1
1	1	1

Para el perceptrón se inicializan w_1 , w_2 y b en valores aleatorios, se evalúan los errores y se corrigen dichos valores hasta que clasifique correctamente

w_1	w_2	b
0.4	0.3	0.2

x_1	x_2	mineral	$\sum(w_i \cdot x_i) + b$	$f(z)$	error
0	0	0	0.2	1	SI
0	1	1	0.5	1	NO
1	0	1	0.6	1	NO
1	1	1	0.9	1	NO

Como en la primera fila hay error, se utilizan estos valores para corregir, utilizando la fórmula de entrenamiento con una tasa de aprendizaje de 0.3.

Aprendizaje	w_{1c}	w_{2c}	b_c
0.3	0	0	-0.3

Vemos que w_1 y w_2 no necesitan corrección y el bias se modifica quedando en -0.1

W1	W2	b			
0.4	0.3	-0.1			
x1	x2	mineral	sum(wi*xi)+ b	f(z)	error
0	0	0	-0.1	0	NO
0	1	1	0.2	1	NO
1	0	1	0.3	1	NO
1	1	1	0.6	1	NO

Finalmente vemos que con la corrección del bias ya se clasifica correctamente todos los casos.

Limitaciones del Perceptrón

- Solo es capaz de resolver problemas linealmente separables. Por ejemplo, puede aprender la función AND o OR, pero no la función XOR.
- Produce salidas binarias, lo que limita su aplicación en problemas de clasificación más complejos.

A pesar de estas limitaciones, el Perceptrón marcó un punto de partida fundamental en el desarrollo de Redes Neuronales Artificiales (RNA). Su concepto de aprendizaje a partir de ejemplos sentó las bases para modelos más avanzados, como las Redes Neuronales Multicapa (MLP), utilizadas en diversas aplicaciones modernas de inteligencia artificial.

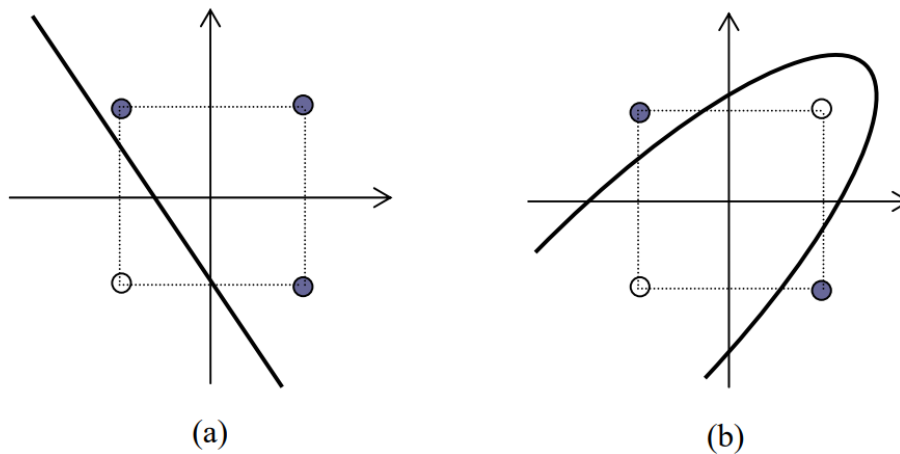


Figura 1. (a) Patrones separables linealmente. (b) Patrones no separables linealmente.

Implementación en python del perceptrón simple

Para comprender el funcionamiento del perceptrón simple, utilizaremos un dataset sintético diseñado para visualizar el desempeño del modelo. En primer lugar, realizaremos una exploración y análisis de los datos para determinar la mejor forma de alimentar el perceptrón, examinando características estadísticas, correlaciones entre variables y otros aspectos relevantes.

Posteriormente, reduciremos la base de datos a la información esencial, separando la variable objetivo de las variables explicativas. Finalmente, utilizaremos el modelo para hacer predicciones.

Como primer paso, importaremos las librerías necesarias (las importadas en el ejemplo anterior), incluyendo el perceptrón de la librería *scikit-learn*, que nos permitirá implementar el modelo de manera eficiente.

```
from sklearn.linear_model import Perceptron
```

Las otras son:

pandas: Manejo de datos en estructuras tipo tabla (DataFrames).

numpy: Operaciones matemáticas y manejo de arreglos numéricos.

matplotlib.pyplot: Creación de gráficos y visualización de datos.

Luego se importa la información, en este caso un Excel llamado dataset_sample.csv, usando pandas:

```
df = pd.read_csv("dataset_sample.csv", sep=";")
df.head()
```

	X	Z	LEY	MINERAL	MINERAL_TEXT
0	51	9	16.252474	0	ESTERIL
1	23	11	16.265250	0	ESTERIL
2	18	8	16.425449	0	ESTERIL
3	60	10	16.565168	0	ESTERIL
4	55	4	16.640253	0	ESTERIL

Se pueden observar las coordenadas X y Z, la ley de cobre, la columna MINERAL (donde 0 representa estéril y 1 indica mineral), y la columna MINERAL_TEXT, que expresa la misma información que MINERAL, pero en formato de texto.

Para comprender mejor la distribución de los datos, se realiza una descripción estadística de ambos grupos (mineral y estéril), agrupándolos y aplicando la función describe().

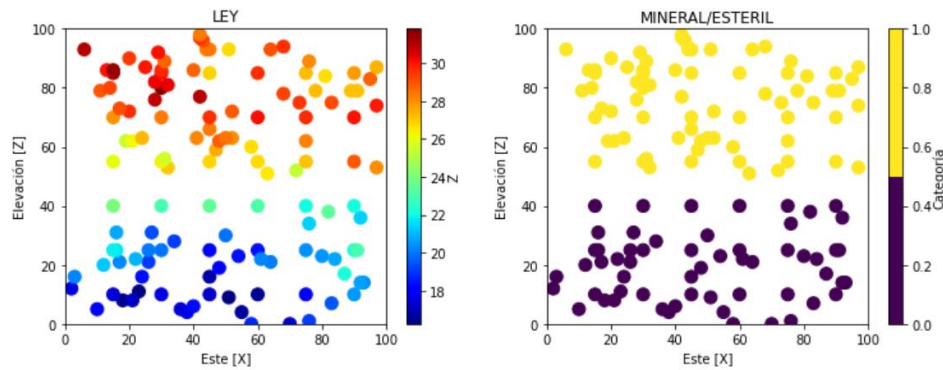
```
group = df.groupby(["MINERAL"]).describe()
group["LEY"]
```

		count	mean	std	min	25%	50%	75%	max
MINERAL									
	0	58.0	19.680602	2.114240	16.252474	17.877996	19.975307	21.196324	24.022730
	1	72.0	28.578024	1.520871	25.427750	27.287337	28.740899	29.654960	31.808874

En estas columnas se pueden observar la cantidad de datos, la desviación estándar, el promedio, los valores mínimos y máximos, así como los cuartiles (el 25%, 50% y 75% de los valores son menores a los respectivos percentiles).

Utilizando matplotlib, se representa gráficamente tanto la distribución del mineral como la ley de cobre en función de las coordenadas X y Z, permitiendo visualizar la variabilidad espacial de estos valores.

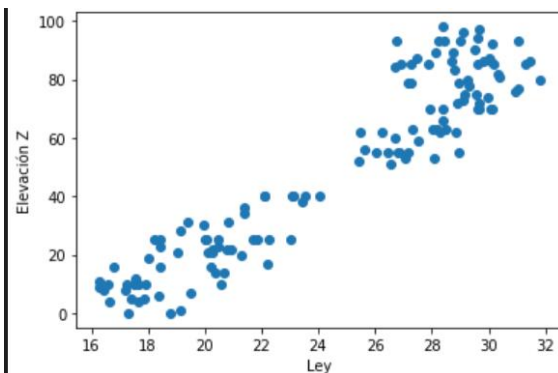
```
marker_size=100
plt.subplot(121)
plt.scatter(df["X"], df["Z"], marker_size, df["LEY"], cmap = plt.cm.jet)
plt.xlabel('Este [X]')
plt.ylabel('Elevación [Z]')
plt.title('LEY')
plt.xlim(0,100)
plt.ylim(0,100)
cbar=plt.colorbar()
cbar.set_label('Z', labelpad=+1)
plt.subplot(122)
plt.scatter(df["X"], df["Z"], marker_size, df["MINERAL"], cmap = plt.cm.get_cmap("viridis", 2))
plt.xlabel('Este [X]')
plt.ylabel('Elevación [Z]')
plt.title('MINERAL/ESTERIL')
plt.xlim(0,100)
plt.ylim(0,100)
cbar=plt.colorbar()
cbar.set_label('Categoría', labelpad=+1)
plt.subplots_adjust(left=0.0, bottom=0.0, right=1.6, top=0.8, wspace=0.2, hspace=0.3)
plt.show()
```



Se observa una correlación espacial entre Z y la presencia de mineral o estéril, donde las leyes más altas se encuentran por encima de $Z = 50$. En cambio, la coordenada X no parece tener una influencia significativa en la distribución de la ley.

Para verificar esta relación, se realiza un scatterplot comparando la ley de cobre con Z, lo que permite analizar visualmente la correlación entre estas variables.

```
plt.scatter(df["LEY"], df["Z"])
#Las variables que se empleen para este modelo deben ser proporcionales guardar alguna relación.
plt.xlabel("Ley")
plt.ylabel("Elevación Z")
plt.show()
```



Como se preveía, existe una correlación entre Z y la ley de cobre, donde la ley tiende a aumentar a medida que Z crece.

Debido a esta relación, se decide utilizar únicamente las variables Z y ley para predecir la etiqueta de mineral o estéril, descartando las demás columnas para simplificar el modelo y enfocarse en los factores más relevantes.

```
df_reduced = df.drop(["X", "MINERAL_TEXT"], axis=1)
df_reduced.head() #Z y Ley serán las variables de entrada, y Mineral será el target.
```

	Z	LEY	MINERAL
0	9	16.252474	0
1	11	16.265250	0
2	8	16.425449	0
3	10	16.565168	0
4	4	16.640253	0

Se separan las columnas explicativas de la columna objetivo.

```
# Separamos las etiquetas de salida del resto de características del conjunto de datos
X_df = df_reduced[["LEY", "Z"]]
y_df = df_reduced["MINERAL"]
```

Se crea el clasificador utilizando el perceptrón, configurado con un máximo de 1000 iteraciones. Aunque este número es excesivo para un caso práctico, se utiliza con fines explicativos (más adelante se abordarán conceptos como overfitting). Además, se establece una semilla en 40 para garantizar la reproducibilidad de los resultados, asegurando que los pesos iniciales se generen de forma aleatoria desde el mismo punto de origen en cada ejecución.

```
clf = Perceptron(max_iter=1000, random_state=40)
clf.fit(X_df, y_df)
```

Se ven los parámetros finales del modelo: los pesos y el bias

```
# Parametros del modelo
clf.coef_

array([[ -237.17031947,  131.          ]])
```

```
# Terminio de interceptacion
clf.intercept_

array([-17.])
```

Quedando la siguiente formula:

$$z(x) = x_1 * w_1 + x_2 * w_2 + b = x_1 * -237.17 + x_2 * 131 + (-17)$$

Para ver el resultado gráficamente, volvemos a hacer el grafico de dispersión, pero dibujando la curva de decisión arrojada por el perceptrón.


```

X = X_df.values

mins = X.min(axis=0) - 0.1
maxs = X.max(axis=0) + 0.1

xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], 1000),
                    np.linspace(mins[1], maxs[1], 1000))

Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

fig = plt.figure(figsize=(10, 7))

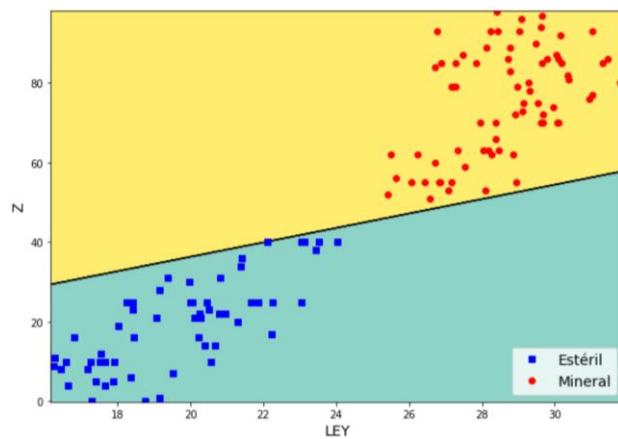
plt.contourf(xx, yy, Z, cmap="Set3")
plt.contour(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]), linewidths=1, colors='k')

plt.plot(X[:, 0][y_df==0], X[:, 1][y_df==0], 'bs', label="Estéril")
plt.plot(X[:, 0][y_df==1], X[:, 1][y_df==1], 'ro', label="Mineral")

plt.xlabel("LEY", fontsize=14)
plt.ylabel("Z", fontsize=14)
plt.legend(loc="lower right", fontsize=14)

plt.show()

```



Y finalmente vemos el desempeño utilizando `accuracy_score`.

```

y_pred = clf.predict(X_df) #en la variable y_pred, se guardan las predicciones del modelo a partir de los datos de entrada en X_df

```

```

accuracy_score(y_df, y_pred) #precisión del 100% del modelo

```

```

1.0

```

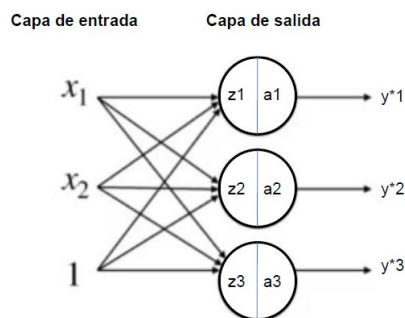
Perceptrón simple monocapa para clasificar múltiples clases

Siguiendo el camino hacia la construcción de una red neuronal artificial, en esta sección se evaluará el uso de un perceptrón simple con múltiples TLUs, para clasificar atributos geológicos como, por ejemplo, tipo de zona mineral.

Arquitectura del Perceptrón y Multi-TLU

El perceptrón básico utiliza una capa de entrada que recibe los datos de las variables explicativas y una capa de salida que genera una clasificación binaria (por ejemplo, mineral o estéril). Su funcionamiento se basa en la combinación lineal de los pesos y las entradas, seguida por una función de activación, como la función Heaviside (función escalón), que determina la salida del modelo:

Para extender esta idea a problemas de clasificación con múltiples categorías, se emplea un Perceptrón Multi-TLU (Threshold Logic Unit), en el cual varias unidades de activación trabajan en paralelo para clasificar instancias en diferentes clases binarias simultáneamente. Esto permite, por ejemplo, diferenciar entre múltiples tipos de mineralización dentro de un depósito.



Función de Activación Softmax

Cuando se requiere una clasificación en múltiples categorías exclusivas (como sulfuros primarios, secundarios y óxidos), se utiliza la función Softmax en la capa de salida. Esta función convierte las activaciones en probabilidades normalizadas entre 0 y 1, asegurando que la suma de las probabilidades de todas las clases sea igual a 1:

$$\text{softmax} = f(z) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}$$

$$y^*1 = \frac{e^{z1}}{e^{z1} + e^{z2} + e^{z3}}$$

$$y^*1 + y^*2 + y^*3 = 1$$

Donde z_i representa la activación de la neurona de salida correspondiente a la clase i . Así, la red neuronal no solo clasifica las muestras, sino que también asigna una probabilidad a cada clase.

Explicación Ejemplo:

Capa de entrada

Se tienen varias características de entrada que representan los datos de entrenamiento. Cada una de ellas se conecta a todas las unidades de salida a través de pesos específicos.

Cálculo de activaciones

Cada unidad de salida combina las entradas mediante una suma ponderada, agregando un valor adicional llamado sesgo. Esto genera un conjunto de valores que servirán para la siguiente etapa.

Función Softmax

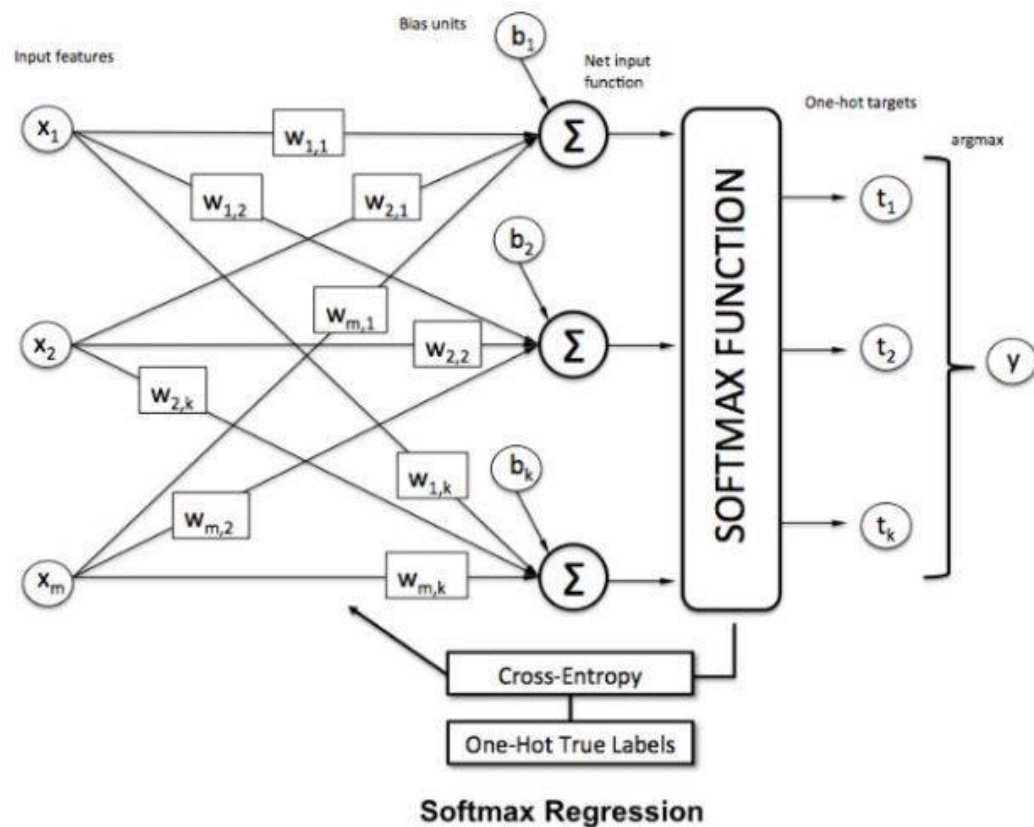
Los valores obtenidos en la etapa anterior se transforman en probabilidades mediante una función que los normaliza. Esto garantiza que la suma total de las probabilidades sea uno, permitiendo que el modelo prediga solo una clase.

One-Hot Encoding y Entropía Cruzada

La salida esperada se representa en un formato donde solo la clase correcta tiene un valor positivo, mientras que las demás son cero, por ejemplo $[0, 1, 0]$. Luego, se mide la diferencia entre la predicción del modelo y la salida real, utilizando una función de error que servirá para ajustar los pesos en el entrenamiento, en este caso se usa Entropía Cruzada (más adelante se explicará cómo funcionan los métodos de optimización).

Predicción final

Se elige la clase con la mayor probabilidad como resultado final del modelo (Argmax de la salida softmax).



Implementación perceptrón simple para clasificar zonificación mineral

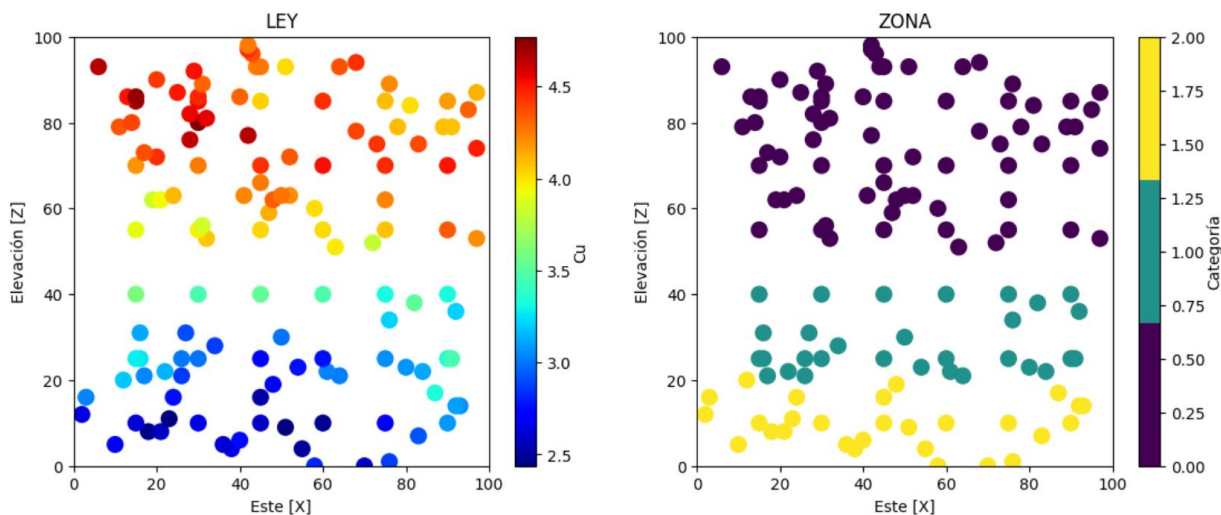
En esta sección veremos implementado en python, como un perceptrón simple con múltiples TLUs puede resolver un problema de clasificación de zonas minerales, utilizando nuevamente un dataset sintético con el fin de implementar el código y evaluar su desempeño.

De manera análoga al ejemplo anterior, primero se importan las librerías y luego usando pandas para abrir y leer el csv, pudiendo ver los primeros datos usando la función `.head()`.

	X	Z	CU	ZONA	ZONA_TEXT
0	70	0	2.592250	2	PRIMARY SULFIDES
1	58	0	2.814284	2	PRIMARY SULFIDES
2	76	1	2.870914	2	PRIMARY SULFIDES
3	55	4	2.496038	2	PRIMARY SULFIDES
4	38	4	2.647656	2	PRIMARY SULFIDES

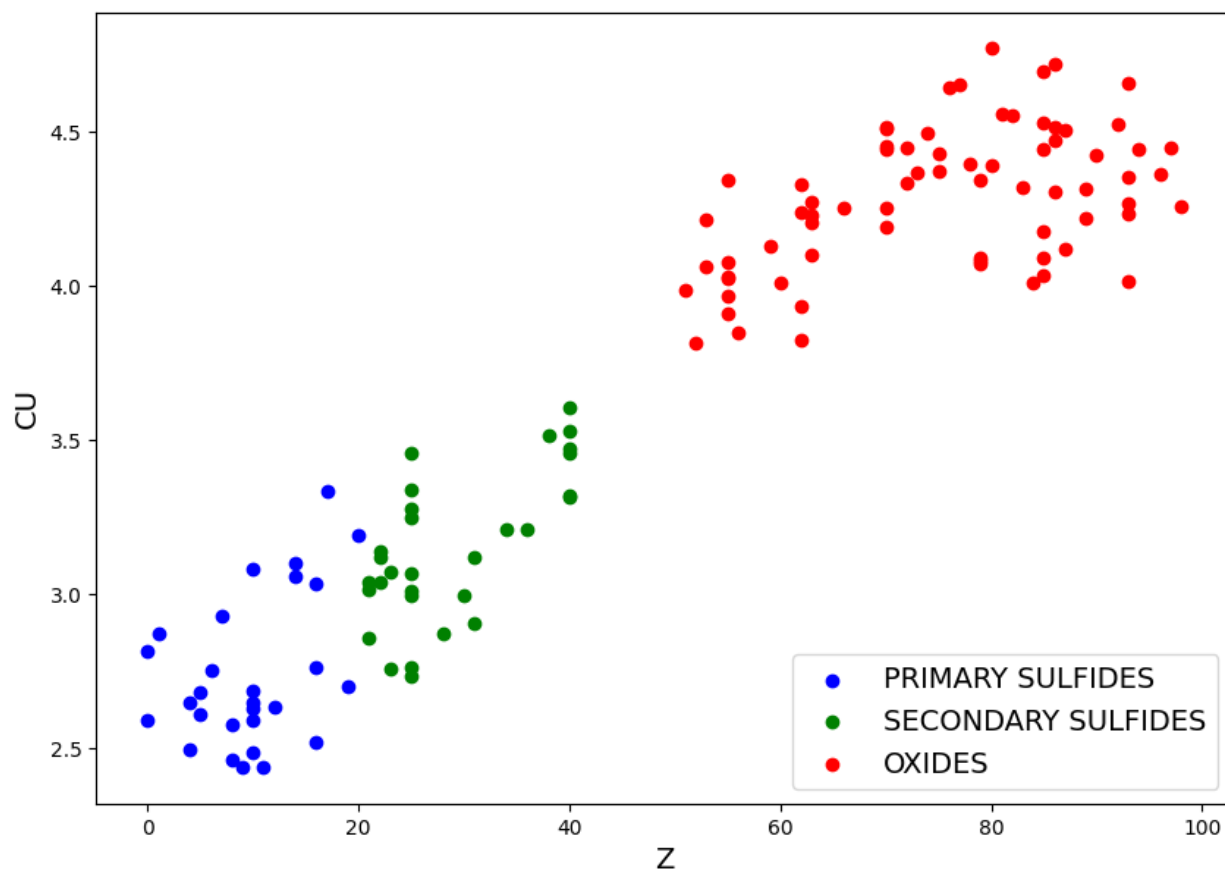
En este dataframe tenemos las coordenadas X y Z, una ley de cobre CU, una zona mineral en la columna ZONA y la zona mineral en texto en la columna ZONA_TEXT.

Utilizando matplotlib, se representa gráficamente tanto la distribución de la zona mineral como la ley de cobre en función de las coordenadas X y Z, permitiendo visualizar la variabilidad espacial de estos valores.



Como se puede apreciar, las leyes tienen una relación directa con la cota Z y con la zona mineral, donde a cotas mas altas hay mayor ley y corresponde a los óxidos.

Se grafica la ley CU en función de la cota Z, diferenciando la zona mineral en distintos colores usando matplotlib. Se excluye la zona mineral en texto y la x por no aportar información.



Nuevamente se observa dicha correlación entre cota y ley más allá de cierta dispersión, quedando los sulfuros primarios como la zona con menos ley y los óxidos con la zona con mejor ley de cobre.

Para observar mejor la distribución estadística de las distintas zonas mineral se agrupa por zona y se utiliza la función `.describe()` con el siguiente resultado, donde podemos ver mínimos, máximos, cantidad de datos, cuartiles, promedio, desviación estándar.

	count	mean	std	min	25%	50%	75%	max
ZONA								
0	72.0	4.286704	0.228131	3.814163	4.093100	4.311135	4.448244	4.771331
1	30.0	3.148601	0.243497	2.735178	3.000168	3.119289	3.317893	3.603410
2	28.0	2.741543	0.244466	2.437871	2.586913	2.665097	2.885211	3.332864

Luego se separa las variables explicativas de las variables objetivo, quedando “Z” y “Q” por un lado y “ZONA” por otro

```
# Separamos las etiquetas de salida del resto de características del conjunto de datos
X_df = df_reduced[["Z", "CU"]]
y_df = df_reduced["ZONA"]
```

Se entrena el clasificador `clf` con un máximo de iteraciones de 1000 y una semilla de 51

```
clf = Perceptron(max_iter=1000, random_state=51) #154645
clf.fit(X_df, y_df)
```

Se evalua el desempeño, el cual fue del 87% de aciertos

```
clf.score(X_df, y_df)
```

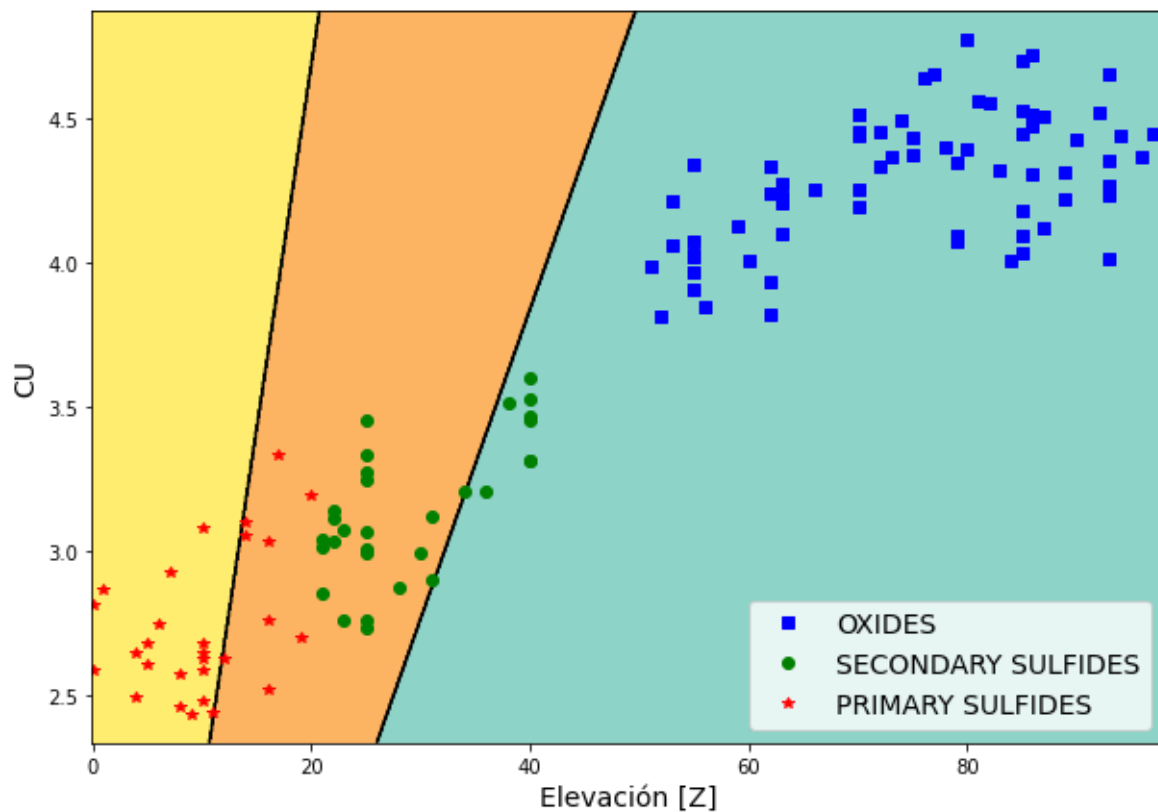
```
0.8692307692307693
```

Vemos los parámetros del modelo, teniendo los pesos de ambas variables para las 3 TLUs.

```
clf.coef_
```

```
array([[ 10.      , -112.03601741],
       [-13.      , 102.82378422],
       [-40.      , 210.08052574]])
```

Finalmente graficamos usando matplotlib, para ver las líneas de separación del modelo.



Como se puede observar, si bien se logra cierta clasificación, presenta dificultades en las fronteras de decisión, donde le falta flexibilidad en las líneas, quedando algunos sulfuros primarios clasificados como secundarios y algunos sulfuros secundarios como óxidos.

Como conclusión este tipo de modelo sirve para aproximar ciertos resultados, pero es muy poco flexible como para poder utilizarlo en casos reales con resultados satisfactorios.

Perceptrón multicapa

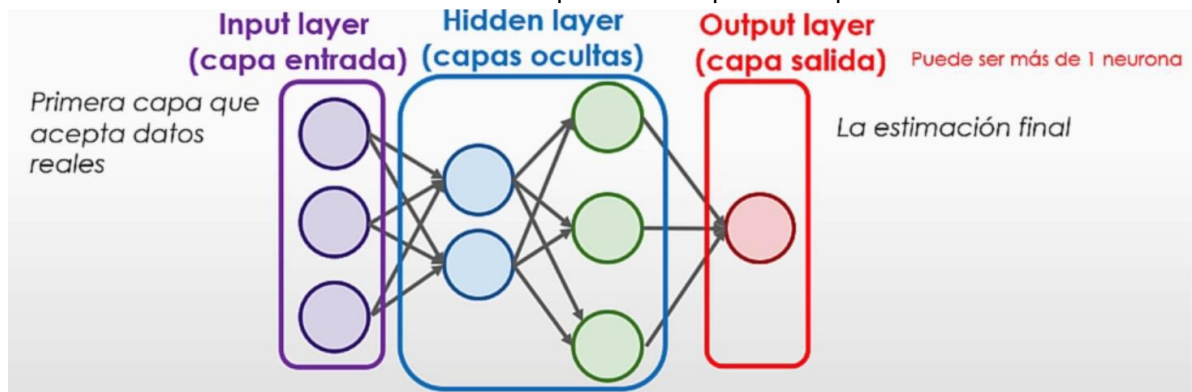
Arquitectura

Debido a las limitaciones del perceptrón simple, surge la necesidad de una arquitectura más avanzada capaz de abordar problemas más complejos, como la clasificación de datos no linealmente separables. El Perceptrón Multicapa (MLP) es una evolución de este modelo que incorpora múltiples capas de neuronas interconectadas, formando lo que se conoce como una red neuronal profunda. Estas redes, ampliamente utilizadas en la actualidad dentro del aprendizaje supervisado, emplean el algoritmo de backpropagation (se explica en profundidad más adelante) para su entrenamiento. Este algoritmo ajusta los pesos de las conexiones neuronales iterativamente, minimizando el error mediante la retropropagación del gradiente, lo que permite mejorar la precisión del modelo y resolver problemas no linealmente separables. Con esta estructura, el MLP amplía la capacidad de modelado y permite representar relaciones más complejas en los datos. En esta sección, se explorará su uso en el contexto de la estimación de leyes y la clasificación aplicada a la generación de modelos geológicos y la definición de dominios.

Arquitectura y Propagación

El Perceptrón Multicapa (MLP) se compone de varias capas de neuronas organizadas en una estructura jerárquica:

- **Capa de entrada:** primera capa de la red, encargada de recibir los datos de entrada. Cada nodo en esta capa representa una característica del conjunto de datos.
- **Capas ocultas:** situadas entre la capa de entrada y la de salida, procesan la información mediante funciones de activación aplicadas a combinaciones ponderadas de las entradas. Si la red cuenta con dos o más capas ocultas, se considera una red neuronal profunda.
- **Capa de salida:** última capa del modelo, cuya función es generar la predicción o clasificación final en función de los valores procesados por las capas ocultas.



En un MLP, los datos fluyen a través de la red en un esquema de propagación hacia adelante (feedforward), avanzando capa por capa hasta llegar a la salida. Dado que todas las capas están completamente conectadas, cada neurona de una capa se enlaza con todas las neuronas de la siguiente, lo que permite una mayor capacidad de representación.

Propagación

Cálculo de la activación en la capa oculta

Cada neurona de la capa oculta recibe una combinación lineal de las entradas, ponderadas por los pesos de las conexiones y sumadas a un sesgo (*bias*).

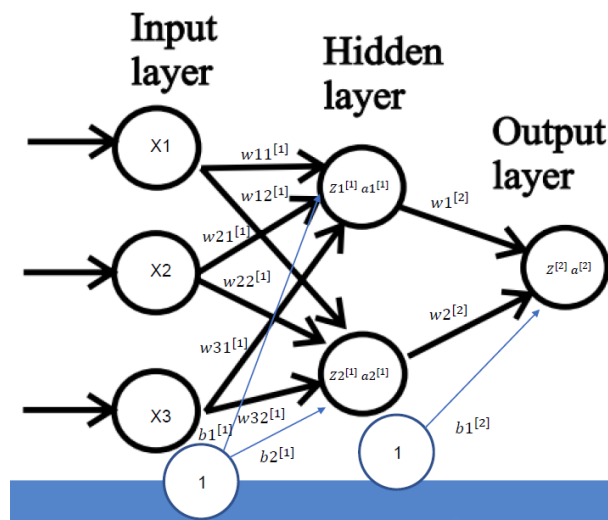
La ecuación general para cada neurona en la capa oculta es:

$$z_j^{(1)} = a_1^{(0)} \cdot w_{1j}^{(1)} + a_2^{(0)} \cdot w_{2j}^{(1)} + a_3^{(0)} \cdot w_{3j}^{(1)} + b_j^{(1)}$$

Donde:

- $z_j^{(1)}$ es la suma ponderada para la neurona j en la capa oculta 1.
- $a_1^{(0)}$, $a_2^{(0)}$, $a_3^{(0)}$ son las entradas de la red, es decir, los valores de X_1 , X_2 y X_3 .
- $w_{ij}^{(1)}$ son los pesos que conectan la neurona i de la capa de entrada con la neurona j de la capa oculta.
- $b_j^{(1)}$ es el sesgo de la neurona j en la capa oculta 1.

Después de calcular $z_j^{(1)}$, se aplica una función de activación $a_j^{(1)} = f(z_j^{(1)})$ para obtener la salida de la neurona en la capa oculta.



Cálculo de la activación en la capa de salida

La salida de la capa oculta se vuelve a combinar en la capa de salida, utilizando otra combinación ponderada:

$$z^{(2)} = a_1^{(1)} \cdot w_1^{(2)} + a_2^{(1)} \cdot w_2^{(2)} + b_1^{(2)}$$

Donde:

- $z^{(2)}$ es la entrada a la neurona de salida.
- $a_1^{(1)}$, $a_2^{(1)}$ son las activaciones de la capa oculta.
- $w_1^{(2)}$, $w_2^{(2)}$ son los pesos de las conexiones entre la capa oculta y la capa de salida.

Al aplicar una función de activación en la capa de salida, obtenemos la predicción final de la red.

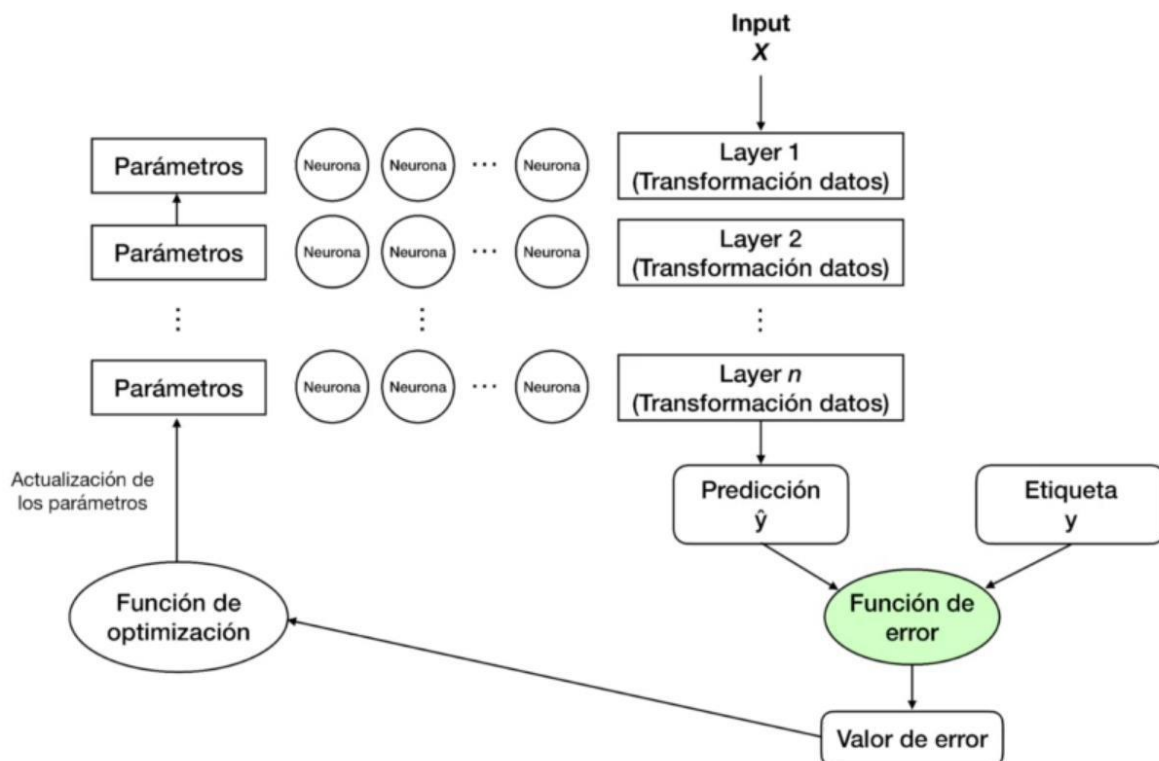
Este es el principio del feedforward, donde la información fluye hacia adelante, sin retroalimentación, desde la entrada hasta la salida.

Luego esa salida deberá ser comparada con la etiqueta real, de esta forma se obtendrá un valor de error.

Función de costo

En el aprendizaje automático, la función de costo (también llamada función de error o función de pérdida) es un componente fundamental que mide la diferencia entre el valor estimado por la red neuronal y el valor real. Su objetivo principal es proporcionar una métrica cuantificable que permita optimizar los parámetros de la red, reduciendo progresivamente el error a lo largo del entrenamiento.

La elección de la función de costo depende del tipo de problema que se desea resolver. A continuación, se describen algunas de las funciones de costo más:



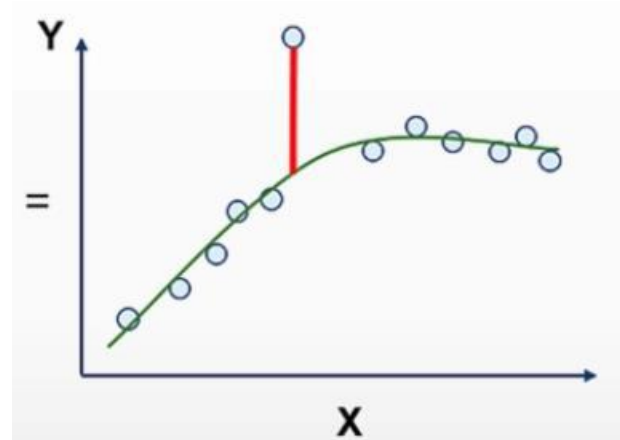
1. Raíz Cuadrada Media (RMSE)

La raíz cuadrada media del error (Root Mean Squared Error, RMSE) es una métrica utilizada en problemas de regresión, especialmente cuando las variables de salida son continuas. Se calcula como la raíz cuadrada del promedio de los errores cuadráticos:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - y_i^*)^2}$$

Donde:

- y es el valor real.
- y^* es el valor predicho por la red.
- n es el número total de observaciones.



Características:

- Penaliza fuertemente los errores grandes, ya que eleva las diferencias al cuadrado.
- Es útil cuando se quiere minimizar la magnitud de los errores de predicción.
- Se utiliza principalmente en modelos de regresión, como por ejemplo la estimación de leyes.

2. Entropía Cruzada Binaria (Binary Cross-Entropy)

La entropía cruzada binaria se emplea en problemas de clasificación binaria, donde la salida es 0 o 1. Su función de costo se define como:

$$J(w) = \frac{-1}{n} \sum_{i=1}^n [y_i \log(y_i^*)] + (1 - y_i) \log(1 - y_i^*)$$

Donde:

- y_i es el valor real (0 o 1) de la muestra i .
- y_i^* es la probabilidad estimada de que la salida sea 1 para la muestra i .
- n es el total de muestras

Características:

- Se utiliza en tareas de clasificación con dos clases.
- Penaliza fuertemente las predicciones erróneas con alta confianza (por ejemplo, si el modelo predice 0.99 y el verdadero valor es 0). Produce valores de pérdida más altos cuando el modelo está muy seguro pero se equivoca, lo que ayuda al aprendizaje.

3. Entropía Cruzada Categórica (Categorical Cross-Entropy)

Cuando el problema de clasificación involucra múltiples clases, se utiliza la entropía cruzada categórica. Su expresión matemática es:

$$J(w) = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m y_{ij} \cdot \log(y_{ij}^*)$$

Donde:

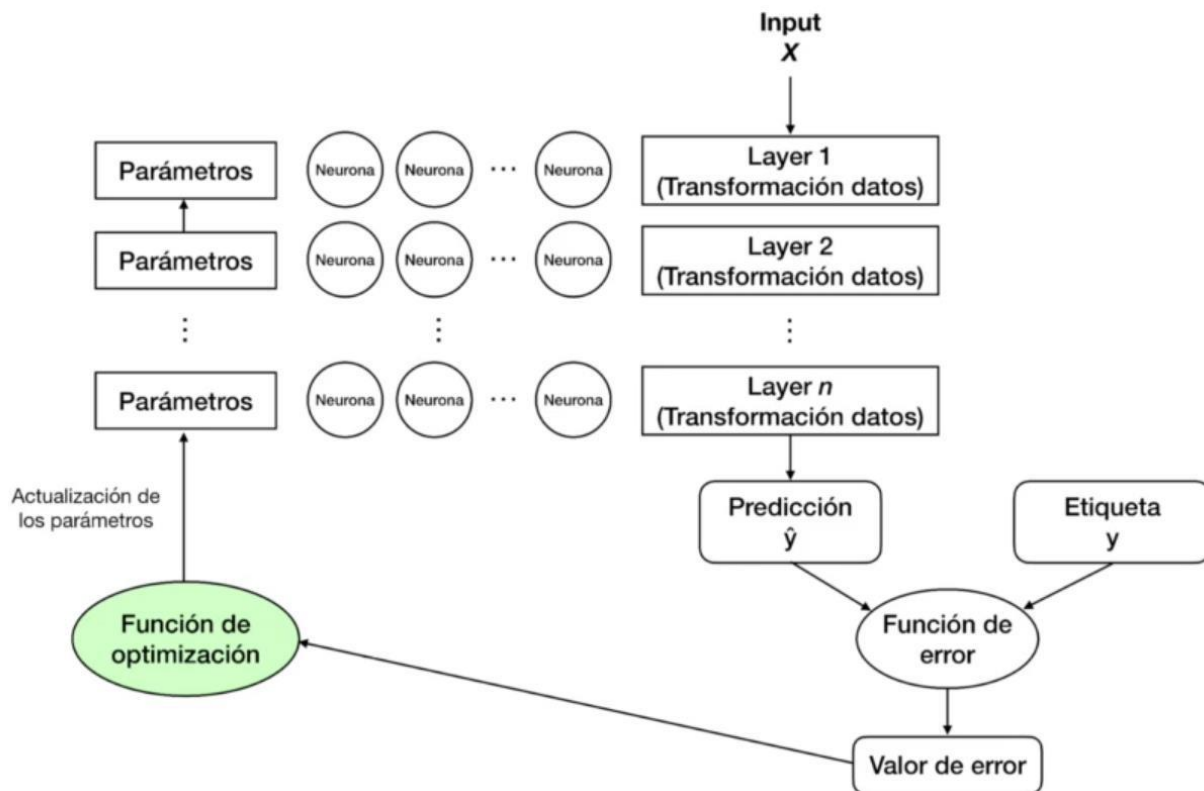
- m es el número de clases.
- y_{ij} es 1 si la muestra pertenece a la clase j , y 0 en caso contrario.
- y_{ij}^* es la probabilidad estimada de que la muestra pertenezca a la clase j .

Características:

- Se emplea en problemas de clasificación multiclase, como por ejemplo identificación de litologías.
- Obliga a la red a asignar probabilidades más altas a la clase correcta.
- Se usa en conjunto con la función de activación *softmax* para obtener probabilidades normalizadas de salida.

En resumen, la función de costo es una pieza clave en el entrenamiento de redes neuronales, ya que guía la actualización de los parámetros a través de algoritmos de optimización como el descenso de gradiente (se explicará más adelante). La elección de una función de costo adecuada depende del tipo de problema que se aborda, asegurando así un aprendizaje más eficiente y preciso.

Función de Optimización



La función de optimización tiene como objetivo minimizar el valor de la función de costo $J(w)$, que mide el error de un modelo de aprendizaje automático. Para lograrlo, se ajustan los parámetros del modelo (pesos w) de manera iterativa hasta encontrar el mínimo de la función de costo.

Descenso por gradiente

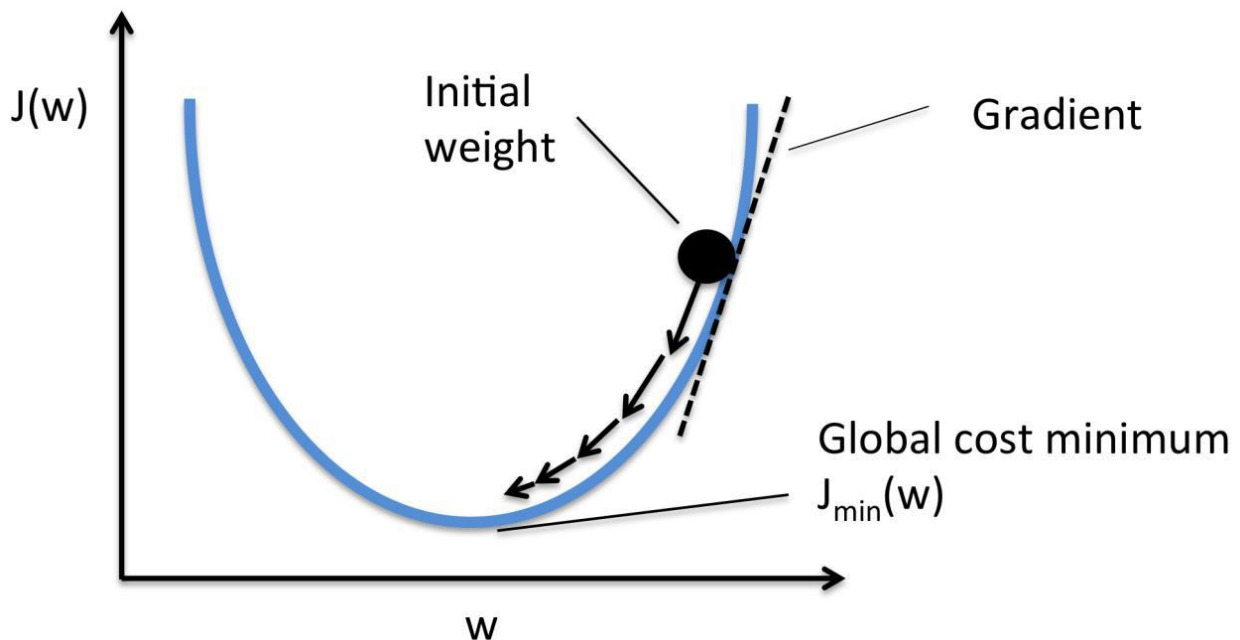
El descenso por gradiente es un algoritmo de optimización que ajusta los pesos de la red neuronal moviéndose en la dirección del gradiente negativo de la función de costo. La actualización de los pesos se realiza con la ecuación:

$$w = w - \eta \frac{dJ}{dw} =$$

Donde:

- w representa los **pesos** del modelo.
- η es el **factor de aprendizaje**, que define el tamaño del paso en cada iteración.
- $\frac{dJ}{dw}$ es la **derivada** de la función de costo con respecto a los pesos, indicando la dirección de la pendiente.

Este proceso se repite hasta alcanzar el **mínimo global de la función de costo** J_{min} .

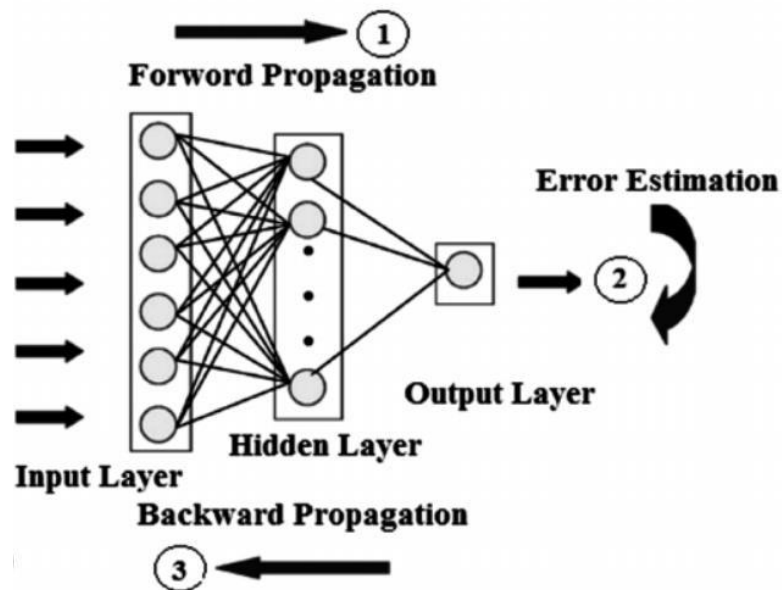


Retropropagación

La retropropagación es el proceso mediante el cual la red ajusta sus parámetros (pesos y sesgos) basándose en el error generado al predecir una salida incorrecta. Este algoritmo se basa en el principio de la *regla de la cadena* y utiliza el *descenso por gradiente* para encontrar los valores de los parámetros que minimizan la función de costo.

En resumen:

- Después de realizar un pase hacia adelante (forward pass), calculando la salida de la red, el algoritmo evalúa el error (función de costo) entre la salida real y la salida deseada.
- Luego, en el pase hacia atrás (backward pass), el algoritmo calcula cómo cada peso contribuye al error y ajusta los pesos y sesgos en consecuencia.
- Este proceso se repite durante varias iteraciones del entrenamiento. En cada ciclo, los pesos y sesgos se ajustan más, permitiendo que la red aprenda gradualmente y reduzca el error en las predicciones.



El objetivo principal es calcular las derivadas de la función de costo con respecto a cada uno de los pesos y sesgos. Este proceso permite determinar cuán "responsable" es cada parámetro (peso) del error final, y así hacer los ajustes necesarios.

La clave de la retropropagación es la regla de la cadena, que permite calcular las derivadas compuestas de manera eficiente. A través de este proceso, el algoritmo puede distribuir el error de la salida hacia los pesos de las capas anteriores, permitiendo que la red aprenda de sus errores en cada capa.

Una vez calculadas todas las derivadas, el ajuste de los pesos se realiza mediante el descenso de gradiente:

$$w_{ij} = w_{ij} - \eta \frac{dl}{dw_{ij}}$$

Donde:

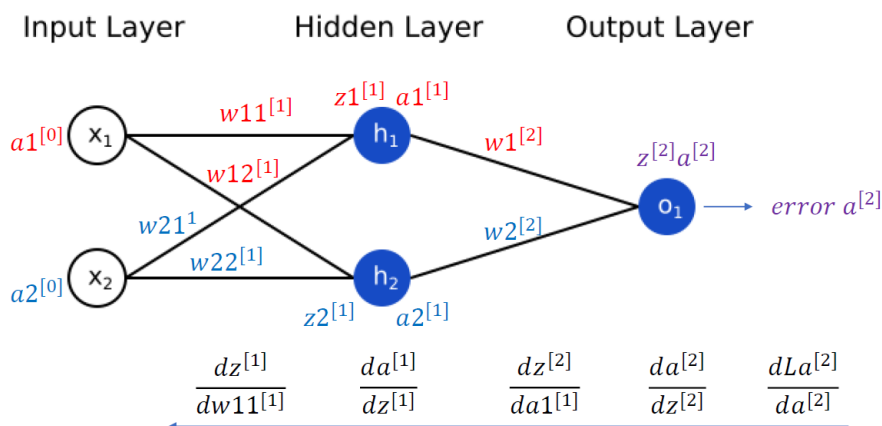
- **w_{ij}** es el peso por ajustar.
- **η** es la tasa de aprendizaje (learning rate), que controla el tamaño del paso.
- **$\frac{dl}{dw_{ij}}$** es la derivada calculada durante la retropropagación.

Resumen:

La retropropagación es un proceso iterativo en el cual, a partir del error obtenido en la salida de la red, se ajustan los parámetros de la red utilizando el cálculo de las derivadas y el descenso por gradiente. El objetivo es minimizar el error de la red ajustando los pesos y sesgos en cada iteración.

Ejemplo 1:

En este ejemplo se busca derivar la función de error ($dL a^{[2]}$) con respecto a uno de los pesos ($w_{11}^{[1]}$). Se omite el uso de bias únicamente por temas prácticos.



En la imagen del ejemplo se ven los pesos w , las sumas ponderadas z , las funciones de activación a y debajo las derivadas que corresponden a cada capa. La fórmula para llegar a la derivada del error respecto al parametro $w_{11}^{[1]}$ se describe a continuación:

$$\frac{dLa^{[2]}}{dw_{11}^{[1]}} = \frac{dLa^{[2]}}{da^{[2]}} * \frac{da^{[2]}}{dz^{[2]}} * \frac{dz^{[2]}}{da_1^{[1]}} * \frac{da^{[1]}}{dz^{[1]}} * \frac{dz^{[1]}}{dw_{11}^{[1]}}$$

Donde:

$\frac{dLa^{[2]}}{da^{[2]}}$ Derivada de la función de costo con respecto a la activación de la segunda capa (salida de la red). Mide cómo el error cambia con respecto a la salida de la red.

$\frac{da^{[2]}}{dz^{[2]}}$ Derivada de la activación $a^{[2]}$ con respecto a la entrada $z^{[2]}$ de la segunda capa. Representa cómo la activación de la segunda capa cambia con respecto a su entrada.

$\frac{dz^{[2]}}{da_1^{[1]}}$ Derivada de la entrada $z^{[2]}$ de la segunda capa con respecto a la activación $a^{[1]}$ de la primera capa. Mide cómo la activación de la primera capa afecta la entrada de la segunda capa.

$\frac{da^{[1]}}{dz^{[1]}}$ Derivada de la activación $a^{[1]}$ con respecto a la entrada $z^{[1]}$ de la primera capa. Indica cómo la activación de la primera capa cambia con respecto a su entrada.

$\frac{dz^{[1]}}{dw_{11}^{[1]}}$ Derivada de la entrada $z^{[1]}$ de la primera capa con respecto al peso $w_{11}^{[1]}$.

Específicamente, muestra cómo cambia la entrada de la primera capa con respecto a ese peso en particular.

Con el resultado de esta derivada, se actualiza el peso $w_{11}^{[1]}$ como se detalló anteriormente de la siguiente manera:

$$w_{ij} = w_{ij} - \eta \frac{dLa^{[2]}}{dw_{11}^{[1]}}$$

Esto se realiza para todos los parámetros, iterativamente hasta llegar al mínimo de la función de error.

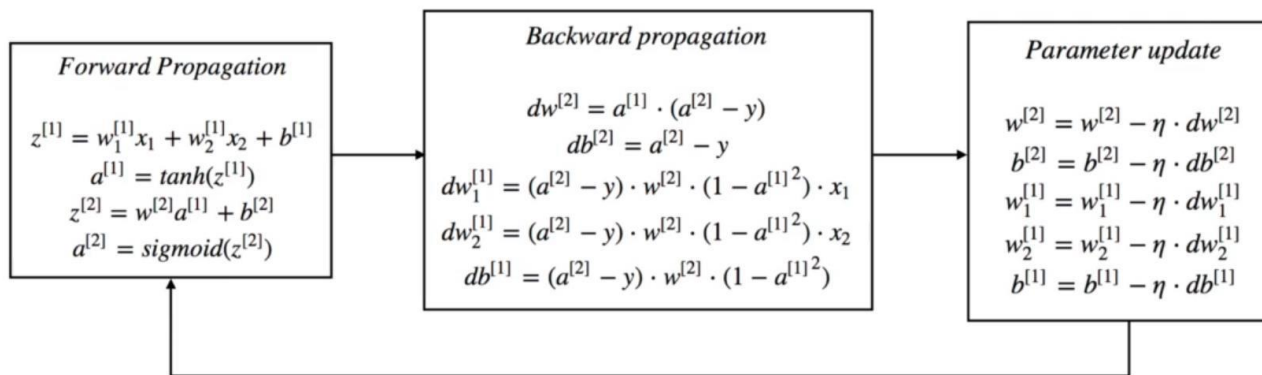
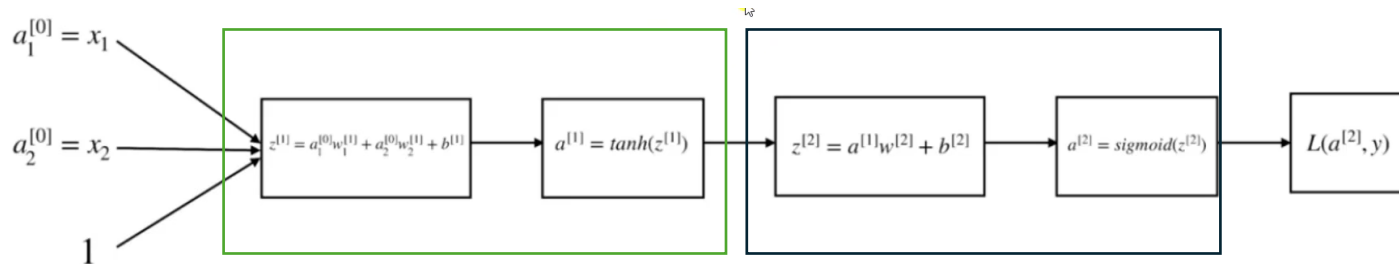
Ejemplo 2:

En este ejemplo, se muestra cómo funciona un ciclo completo de retropropagación para una red neuronal simple con dos entradas y dos neuronas. En la primera capa, se utiliza una neurona con la función de activación tangente hiperbólica (\tanh) y en la segunda capa, una neurona con la función de activación sigmoide (σ).

Parámetros a optimizar:

- **Pesos:** w_1 y w_2 en la primera capa, y w en la segunda capa.
- **Sesgos:** Hay un sesgo para cada capa.

En total, hay 5 parámetros a optimizar: w_1 , w_2 , w y los dos sesgos (uno para cada capa).



La segunda imagen está dividida en tres partes principales:

Forward Propagation:

- En esta parte se muestra el cálculo hacia adelante, donde las entradas se pasan a través de las funciones de agregación y luego se aplican las funciones de activación.
 - **Primera capa:** Se utiliza $\tanh(z_1)$ para la primera neurona.
 - **Segunda capa:** Se utiliza $\text{sigmoide}(z_2)$ para la segunda neurona, produciendo la salida final.

Cálculo de Derivadas Parciales:

- En esta parte se calculan las derivadas parciales de la función de costo con respecto a cada uno de los 5 parámetros (pesos y sesgos) usando la regla de la cadena. Esto muestra cómo cada parámetro influye en el error final.

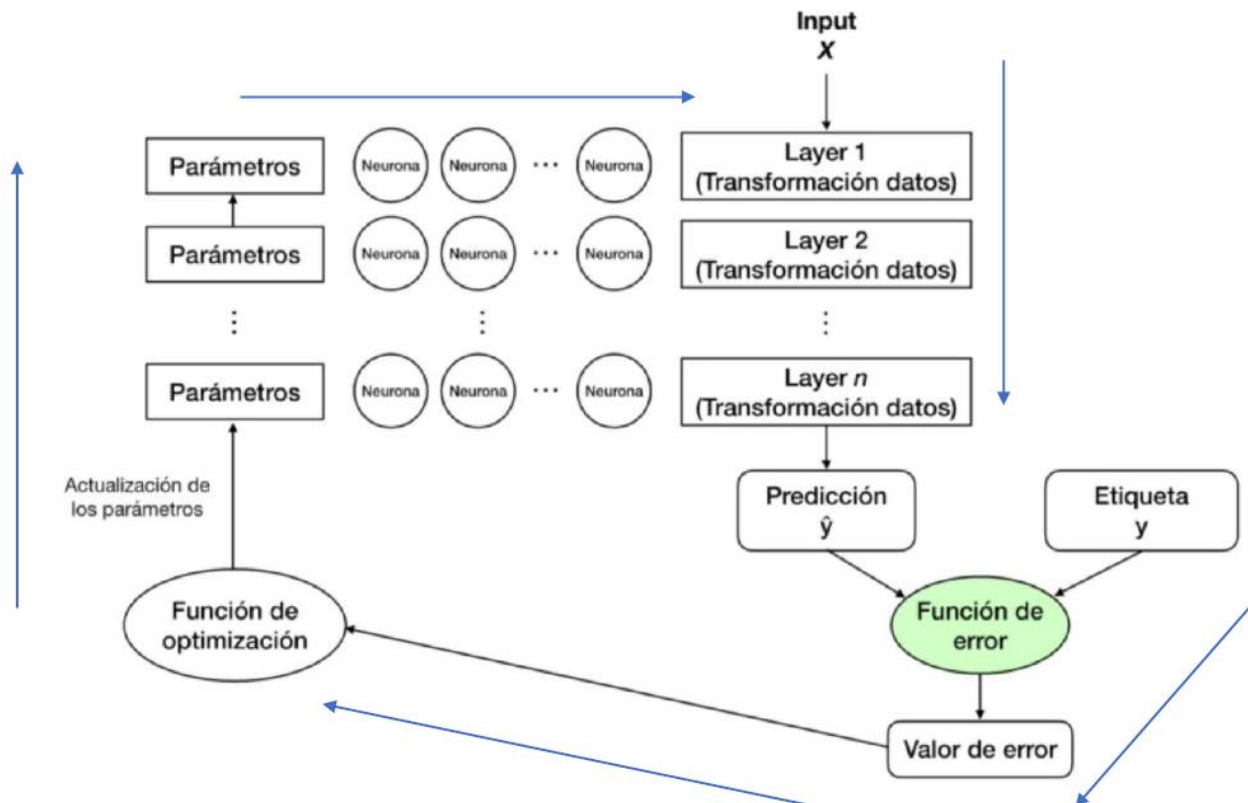
Actualización de Parámetros:

- En esta sección, se muestra cómo los parámetros (pesos y sesgos) se actualizan mediante el descenso por gradiente. Usando las derivadas parciales obtenidas en el paso anterior, los parámetros se ajustan para minimizar el error de la red.

Resumen Perceptrón multicapa:

De esta manera, podemos comprender el ciclo completo de una red neuronal de perceptrón multicapa. Este ciclo comienza con la entrada de las variables, pasa por el proceso de cálculo de la salida, y luego se compara con la salida esperada. A continuación, se calcula el error mediante la función de costo. La función de optimización toma los parámetros del modelo (pesos y sesgos) y, mediante el proceso de retropropagación, los ajusta para minimizar el error.

Este ciclo se repite en cada iteración, actualizando los pesos de la red, lo que genera un nuevo error en cada paso. El proceso continúa hasta que los parámetros del modelo convergen, alcanzando el valor final que minimiza el error, lo que optimiza el rendimiento del modelo.



Implementación del Perceptrón multicapa en python

En este caso, se retoma el dataset sintético utilizado con el perceptrón simple y se realizan las mismas pruebas para corroborar la mejora en la separación utilizando el perceptrón multicapa.

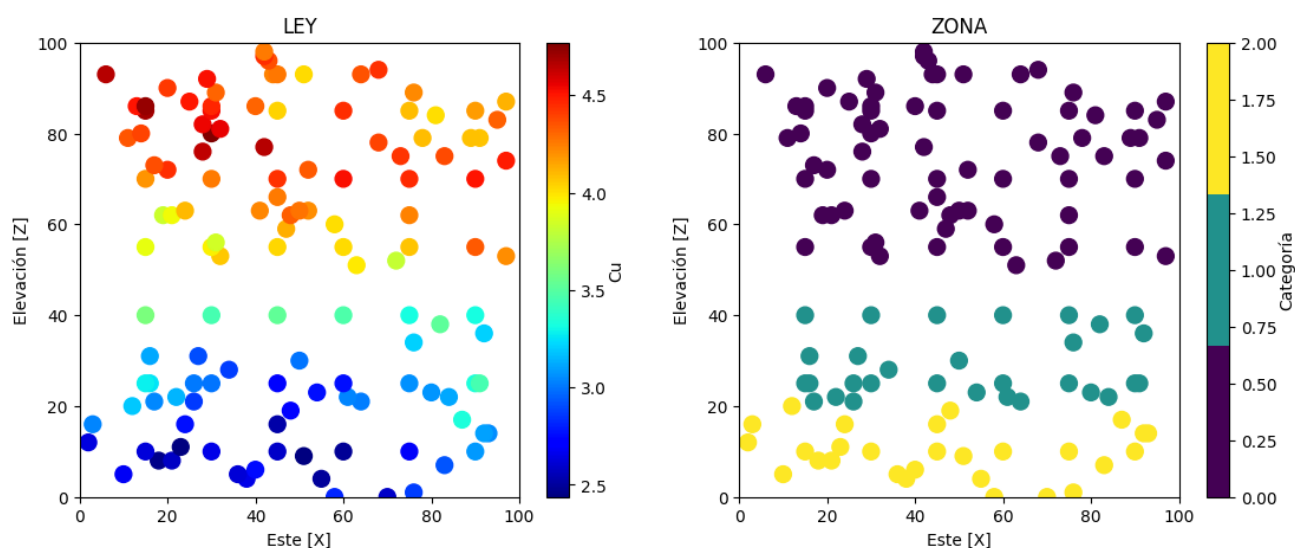
El código es similar al empleado anteriormente, pero con diferencias en la forma de entrenar el modelo, como se verá a continuación.

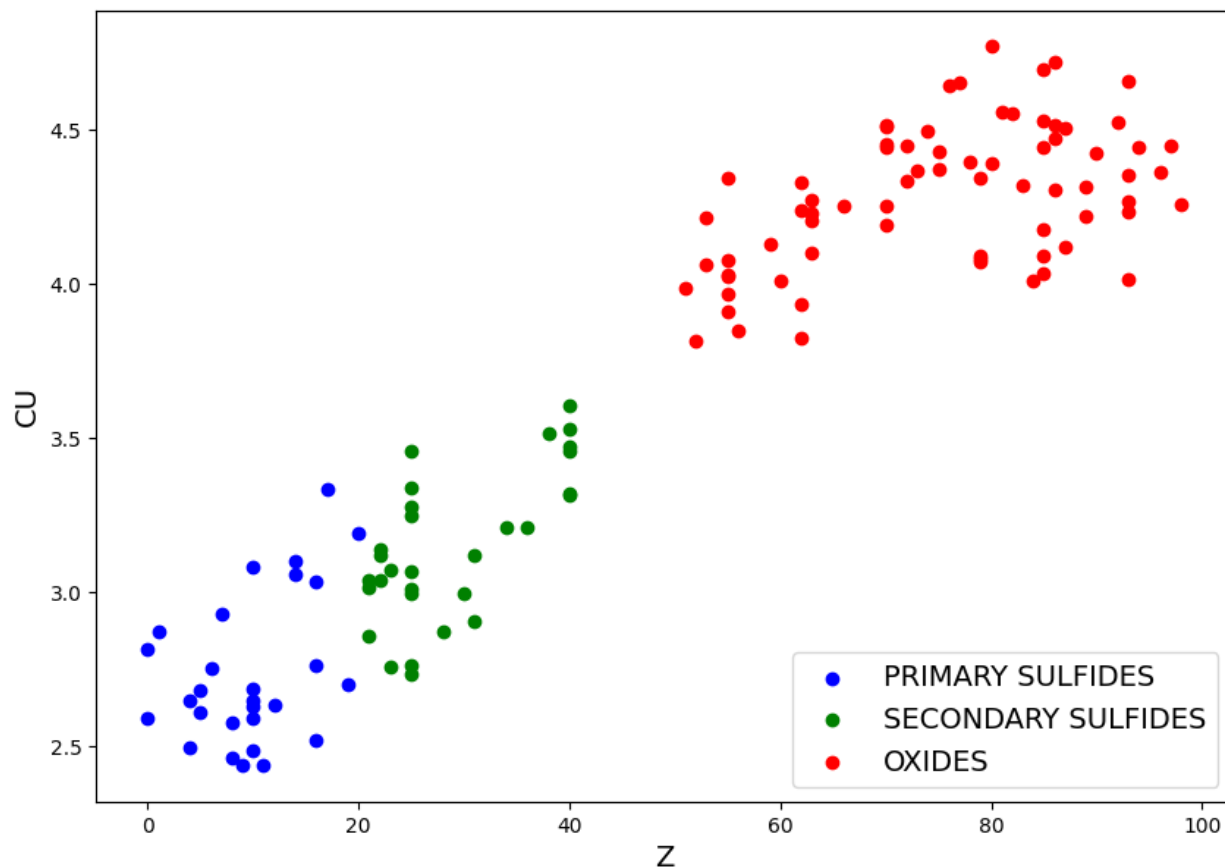
El primer paso consiste en importar las librerías necesarias, abrir el archivo y explorar los datos. Para recordar, el archivo contiene las siguientes variables:

	X	Z	CU	ZONA	ZONA_TEXT
0	70	0	2.592250	2	PRIMARY SULFIDES
1	58	0	2.814284	2	PRIMARY SULFIDES
2	76	1	2.870914	2	PRIMARY SULFIDES
3	55	4	2.496038	2	PRIMARY SULFIDES
4	38	4	2.647656	2	PRIMARY SULFIDES

En el conjunto de datos se pueden observar las coordenadas X y Z, la ley de cobre, la columna "MINERAL" (donde 0 representa estéril y 1 indica mineral), y la columna "MINERAL_TEXT", que expresa la misma información que "MINERAL", pero en formato de texto.

Es importante recordar la distribución espacial, donde se puede visualizar la correlación entre el cobre, la ley y el mineral.





Al igual que con el perceptrón simple, se decide utilizar únicamente las variables Z y la ley para predecir la etiqueta de mineral, descartando las demás columnas para simplificar el modelo y centrarse en los factores más relevantes.

Se separan las etiquetas de salida del resto de las características del conjunto de datos y se procede a entrenar el modelo utilizando el siguiente código:

```
from sklearn.neural_network import MLPClassifier

clf = MLPClassifier(hidden_layer_sizes=(120,60,30), activation='logistic',max_iter=600, random_state=15678151)
clf.fit(X_train, y_train)
```

✓ 5.5s

MLPClassifier

MLPClassifier(activation='logistic', hidden_layer_sizes=(120, 60, 30), max_iter=600, random_state=15678151)

En este código se entrena un clasificador de red neuronal utilizando el perceptrón multicapa (MLPClassifier) de la librería sklearn.neural_network. A continuación, se detalla cada una de las partes del código:

Importación de la librería:

Se importa el modelo MLPClassifier de la librería sklearn.neural_network, el cual permite crear una red neuronal de perceptrón multicapa.

Creación del clasificador:

El clasificador MLPClassifier se inicializa con ciertos parámetros:

- **hidden_layer_sizes=(120, 60, 30):** Define la estructura de la red neuronal especificando el número de neuronas en cada capa oculta. En este caso, se tienen tres capas ocultas con 120, 60 y 30 neuronas, respectivamente.
- **activation='logistic':** La función de activación utilizada en las neuronas de la red es la función sigmoide (o logística). Aunque esta función es comúnmente utilizada en problemas de clasificación binaria, en este caso también puede ser utilizada para problemas de clasificación multiclase a través del enfoque one-vs-rest (uno contra el resto), donde se entrenan clasificadores binarios para cada clase.
- **max_iter=600:** Define el número máximo de iteraciones que realizará el modelo durante el entrenamiento. El modelo intentará ajustarse en 600 pasos como máximo, deteniéndose antes si se encuentra una solución adecuada.
- **random_state=15678151:** Establece una semilla aleatoria para asegurar que los resultados sean reproducibles en cada ejecución del código.

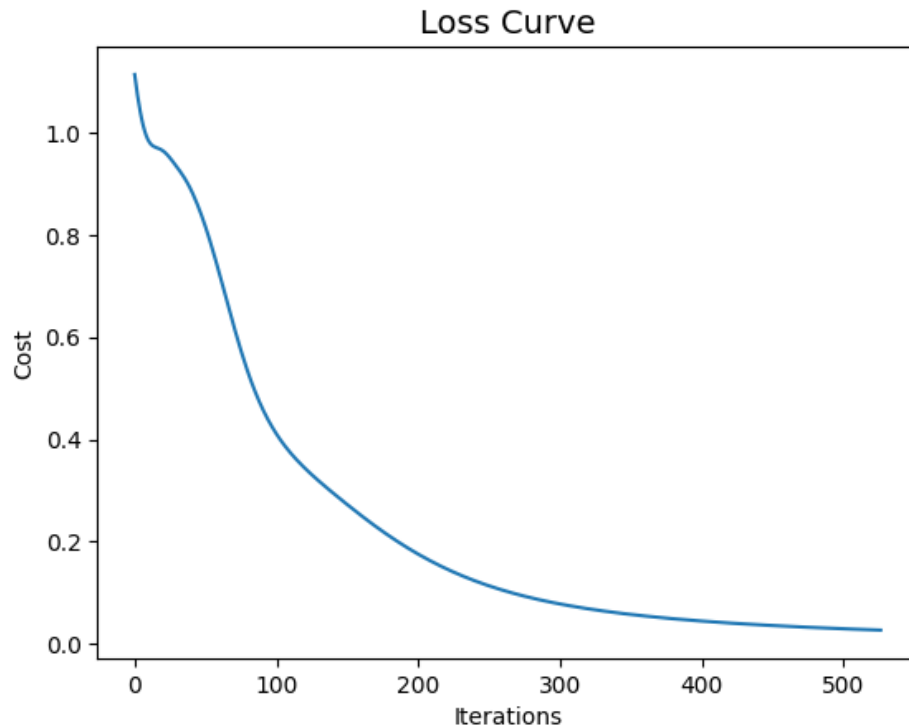
Entrenamiento del modelo:

La función fit() se utiliza para entrenar el modelo utilizando los datos de entrenamiento (X_Train para las características de entrada y Y_train para las etiquetas de salida). En este paso, el modelo ajusta sus parámetros internos (pesos y sesgos) para minimizar el error en las predicciones y aprender a predecir correctamente las etiquetas asociadas a las características de entrada. En este caso, las etiquetas a predecir son de tipo categórico multiclase: *primary sulfides*, *secondary sulfides* y *oxides*, por lo que el modelo realizará una clasificación multiclase.

En este caso, no se pasa explícitamente ningún parámetro para la función de optimización. Al omitir este parámetro, el modelo utilizará el optimizador por defecto de MLPClassifier, que es Adam. Este optimizador será explicado en detalle en la siguiente sección.

Para ver la evolución de la función de costo durante el entrenamiento del modelo, se ejecuta el siguiente código con el gráfico adjunto como resultado.

```
plt.plot(clf.loss_curve_)
plt.title("Loss Curve", fontsize=14)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.show()
```



Se observa como la curva decrece rápidamente en los primeros ciclos, llegando poco a poco al mínimo de la función de costo.

Explicación línea por línea:

- `plt.plot(clf.loss_curve_)`: Grafica la curva de pérdida (`loss_curve_`), que muestra cómo disminuye el error a lo largo de las iteraciones.
- `plt.title("Loss Curve", fontsize=14)`: Asigna el título "**Loss Curve**" a la gráfica con un tamaño de fuente de 14.
- `plt.xlabel('Iterations')`: Etiqueta el eje X como "**Iterations**", representando el número de ciclos de entrenamiento.
- `plt.ylabel('Cost')`: Etiqueta el eje Y como "**Cost**", indicando el valor de la función de costo.
- `plt.show()`: Muestra la gráfica.

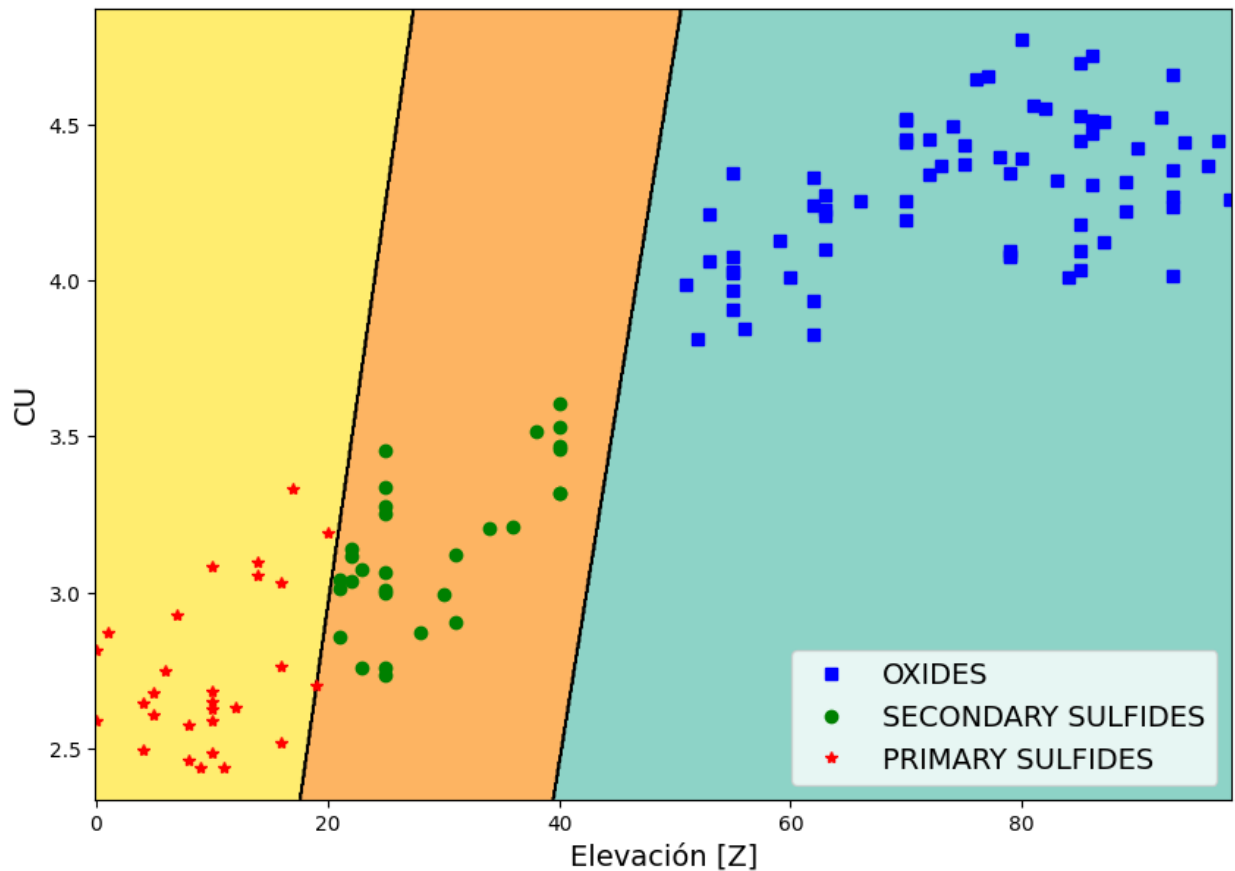
Esta curva es útil para evaluar si el modelo está convergiendo correctamente o si es necesario ajustar hiperparámetros como la tasa de aprendizaje o el número de iteraciones.

Se evalúa el resultado:

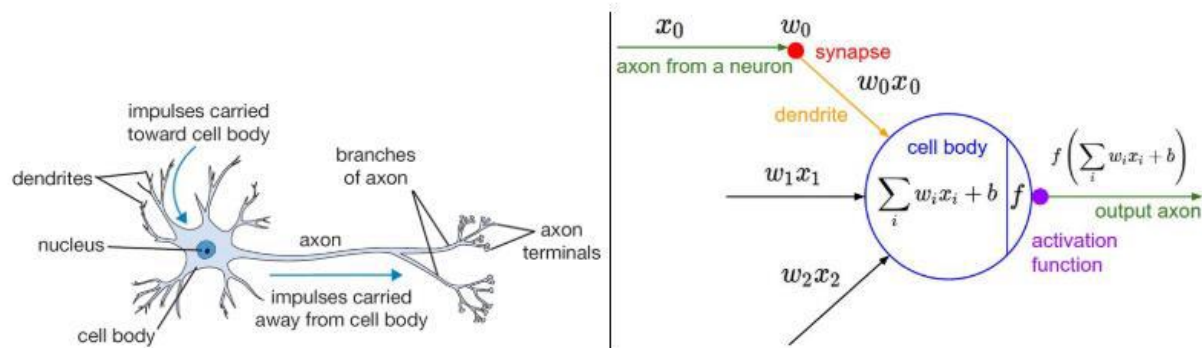
```
clf.score(X_train,y_train)
✓ 0.0s
1.0
```

Si bien lo ideal sería dividir los datos en conjuntos de entrenamiento, validación y prueba, como se verá más adelante, en este caso, debido al tamaño reducido del dataset, la evaluación se realiza directamente sobre el conjunto de entrenamiento. Esto arroja un 100% de precisión, evidenciando una mejora significativa en comparación con el perceptrón simple.

Para visualizar las líneas de decisión del modelo, se genera un gráfico de manera similar al utilizado con el perceptrón simple, obteniendo el siguiente resultado:



Función de Activación en Redes Neuronales Artificiales



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Una función de activación es una función matemática de la forma que se incorpora en una red neuronal artificial con el objetivo de permitir que la red aprenda patrones complejos en los datos. Su importancia radica en que introduce no linealidad en la red, lo que le permite modelar relaciones no lineales entre las variables de entrada y salida.

Importancia de la No Linealidad

Si una red neuronal no contara con funciones de activación no lineales, cada capa de la red sería simplemente una combinación lineal de la anterior. En consecuencia, sin importar cuántas capas se agreguen, la red seguiría representando una transformación lineal de la entrada, limitando drásticamente su capacidad para aprender estructuras complejas. La función de activación permite que la red aprenda representaciones más abstractas y complejas de los datos.

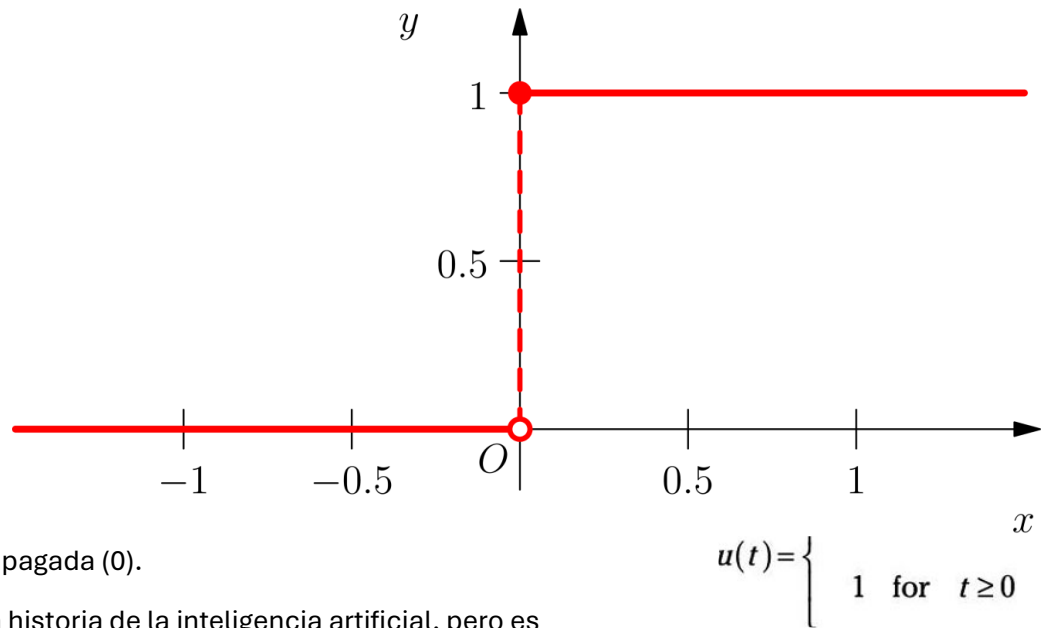
Derivabilidad de la Función de Activación

Para que una red neuronal pueda ser entrenada de manera eficiente mediante algoritmos de optimización como el descenso de gradiente, es crucial que la función de activación sea derivable. La derivabilidad permite calcular la retropropagación del error, ajustando los pesos de la red en cada iteración del entrenamiento. En este sentido, algunas funciones de activación como la sigmoide, la tangente hiperbólica y la ReLU tienen derivadas bien definidas que facilitan la actualización de los parámetros de la red.

A continuación, se presentan algunas funciones de activación comunes utilizadas en redes neuronales artificiales:

Función Escalón

La función escalón de Heaviside, también conocida como función escalón unitario, es discontinua: toma el valor 0 para entradas negativas y 1 para entradas positivas o cero. Fue introducida en redes neuronales por McCulloch y Pitts en 1943, como parte de su modelo de neuronas binarias, donde cada neurona solo podía estar encendida (1) o apagada (0).



Este modelo fue un hito en la historia de la inteligencia artificial, pero es excesivamente simplista. Las neuronas reales no funcionan de forma estrictamente binaria; su activación varía en un rango continuo, lo que permite mayor flexibilidad en la representación de información.

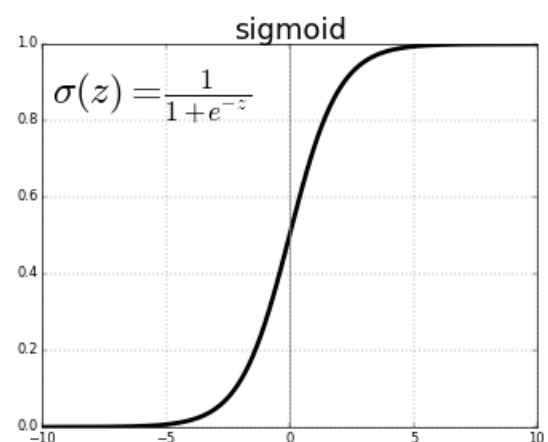
Además, la función escalón **no es derivable**, lo que impide su uso en algoritmos de aprendizaje modernos como el descenso de gradiente. Por ello, hoy en día se utilizan funciones de activación como **sigmoide**, **ReLU** o **softmax**, que permiten ajustes graduales de los pesos y mejor desempeño en tareas complejas.

Función Sigmoide

La función sigmoide transforma cualquier valor real en un número entre 0 y 1, lo que la hace útil para **clasificación binaria**, especialmente en la **capa de salida** de redes neuronales.

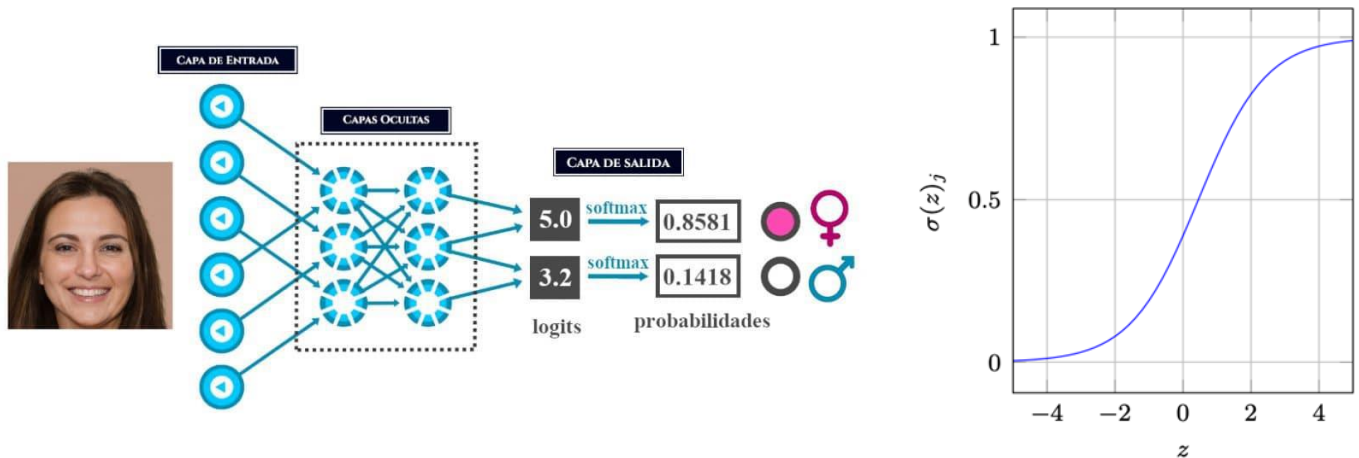
Características:

- **Rango:** (0, 1)
- **Centro:** $\sigma(0) = 0.5$
- **Asíntotas:** Tiende a 0 en $-\infty$ y a 1 en ∞
- **Derivable y continua:** Ideal para descenso de gradiente



Aunque es muy usada en capas ocultas, hoy suele ser reemplazada por ReLU en redes profundas. Su principal desventaja es la saturación de gradientes en valores extremos, lo que puede dificultar el aprendizaje.

Función Softmax



La función softmax se usa en la capa de salida para problemas de clasificación multiclase. Convierte un vector de valores en una distribución de probabilidad, donde la suma de todas las salidas es 1.

Características:

- **Formula:**
$$f(z) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}$$
- **Rango:** Cada valor está entre 0 y 1, y la suma total es 1
- **Continua y derivable:** Compatible con descenso de gradiente
- **Normalización:** Cada salida representa la probabilidad relativa de una clase
- **Uso exclusivo:** En salidas de modelos multiclase
- **Predicción:** Se elige la clase con mayor probabilidad

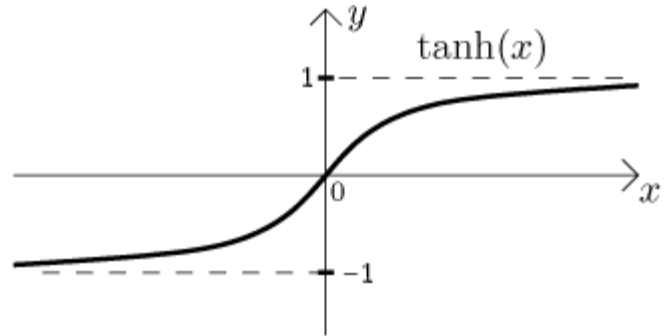
Aunque puede sufrir saturación de gradientes, sigue siendo esencial cuando se requiere interpretar salidas como probabilidades.

Función Tangente Hiperbólica (tanh)

La función tanh transforma los valores de entrada al rango $(-1, 1)$, lo que la hace útil en capas ocultas donde se requiere representar tanto valores positivos como negativos.

Características:

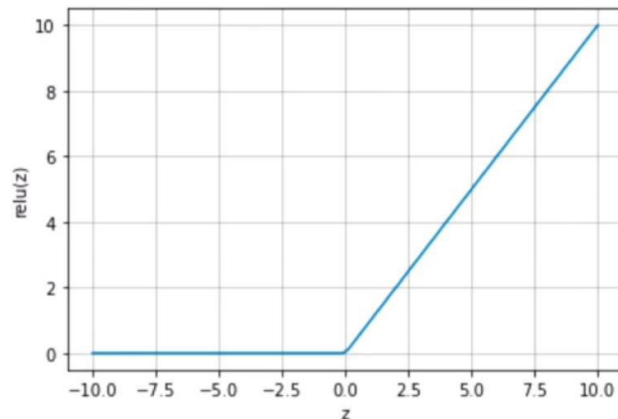
- **Formula:** $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- **Rango:** $(-1, 1)$
- **Centro en cero:** Mejora la convergencia frente a la sigmoide
- **Continua y derivable**
- **Asíntotas:** Tiende a -1 en $-\infty$ y a 1 en ∞
- **Limitación:** Puede saturarse en valores extremos y es más costosa computacionalmente que ReLU.



Sigue siendo útil en redes donde el centrado en cero es importante, a pesar de sus limitaciones.

Función ReLU (Rectified Linear Unit)

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



ReLU es una de las funciones de activación más usadas en redes neuronales profundas, por su simplicidad y eficiencia.

Características:

- **Rango:** $[0, \infty)$
- **Continua**, pero no derivable en $x = 0$ (se define 0 o 1 por convención)
- **Comportamiento:**
 - $x \leq 0 \rightarrow$ salida 0 (neurona inactiva)
 - $x > 0 \rightarrow$ salida x (lineal)

- **Uso:** Común en capas ocultas, por evitar saturación de gradientes
- **Compatibilidad:**
 - Con **sigmoide** en salidas binarias
 - Con **softmax** en salidas multiclase

Ventajas:

- Evita desaparición del gradiente
- Muy eficiente computacionalmente

Limitaciones:

- No derivable en 0
- Puede generar **neuronas muertas** (quedan en 0 permanentemente)

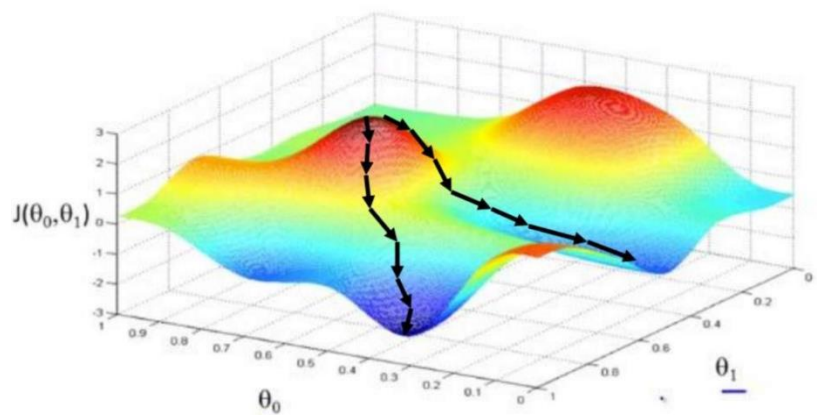
Funciones de optimización

Como mencioné anteriormente, el entrenamiento de una Red Neuronal Artificial (RNA) es un proceso iterativo en el que se ajustan los parámetros del modelo, como los pesos y los bias, con el objetivo de minimizar el error en las predicciones. Este error se cuantifica mediante una función de costo, la cual mide la diferencia entre las salidas esperadas y las obtenidas por la red. Para reducir dicho error y mejorar el desempeño del modelo, se emplean técnicas de optimización que ajustan los parámetros de manera eficiente.

Las funciones de optimización son algoritmos diseñados para guiar la actualización de los pesos de la RNA en cada iteración del entrenamiento. La clave de estas funciones es encontrar un equilibrio entre la rapidez de convergencia y la estabilidad del modelo. Un buen optimizador debe ser capaz de reducir el error sin caer en mínimos locales indeseados ni generar oscilaciones que impidan alcanzar una solución óptima.

El problema de optimización puede visualizarse como el proceso de encontrar el punto más bajo en un paisaje tridimensional donde la altura representa el error del modelo. Para descender en esta superficie y encontrar el mínimo global, los algoritmos de optimización utilizan el gradiente de la función de costo, que indica la dirección en la que el error

disminuye más rápidamente. La magnitud del paso dado en esa dirección está determinada por la



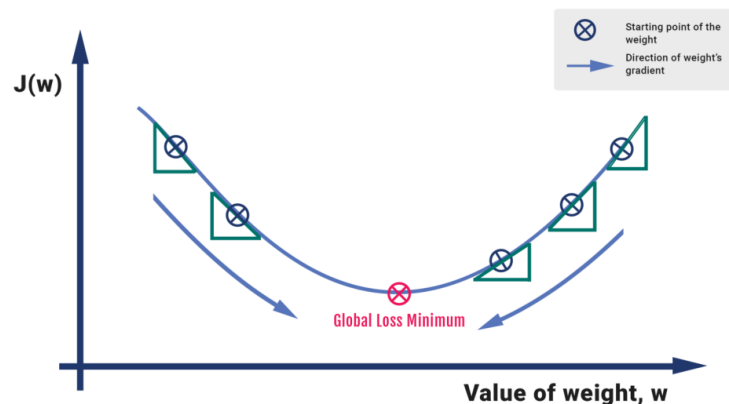
tasa de aprendizaje, un hiperparámetro crucial que influye en la velocidad y precisión del entrenamiento.

Existen diversos métodos de optimización que han sido desarrollados para mejorar la eficiencia y precisión del aprendizaje en redes neuronales. Entre ellos se encuentran el ya mencionado descenso por gradiente, que es la base fundamental de muchos otros algoritmos; la variante del descenso por gradiente estocástico (SGD), que introduce aleatoriedad para mejorar la generalización del modelo; y optimizadores avanzados como Adam, RMSprop, que ajustan dinámicamente la tasa de aprendizaje para mejorar la estabilidad y velocidad de convergencia.

A continuación, exploraremos en detalle las principales funciones de optimización utilizadas en redes neuronales artificiales.

Descenso por Gradiente

El descenso por gradiente es uno de los algoritmos de optimización más utilizados en el entrenamiento de redes neuronales artificiales. Su objetivo es minimizar la función de costo ajustando iterativamente los parámetros del modelo (pesos y bias) en la dirección del gradiente negativo de la función de costo.



➤ Funcionamiento

El descenso por gradiente sigue una serie de pasos fundamentales:

1. Inicialización de los parámetros del modelo con valores aleatorios o determinados previamente.
2. Cálculo del error mediante la función de costo.
3. Cálculo del gradiente, es decir, la derivada parcial de la función de costo con respecto a cada uno de los parámetros del modelo.
4. Actualización de los parámetros utilizando la siguiente fórmula:

$$w = w - \eta \frac{dJ}{dw} =$$

Donde:

w representa los pesos del modelo.

η es el factor de aprendizaje, que define el tamaño del paso en cada iteración.

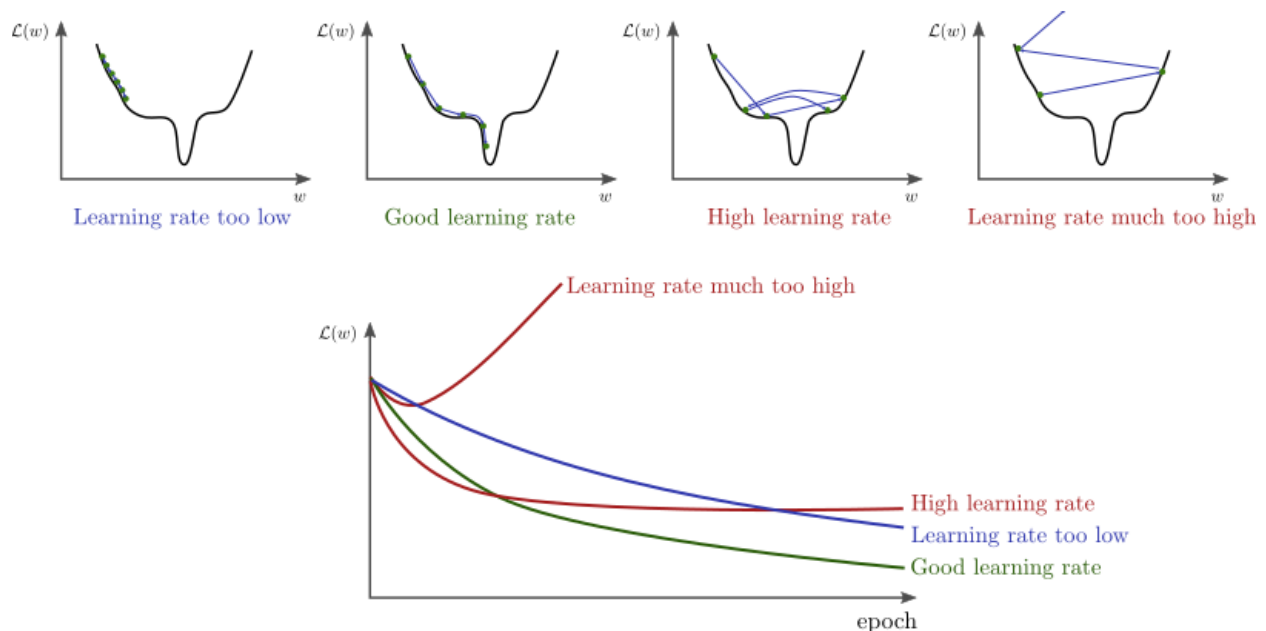
$\frac{dJ}{dw}$ es la derivada de la función de costo con respecto a los pesos, indicando la dirección de la pendiente.

5. Repetición del proceso hasta que la función de costo alcance un mínimo satisfactorio.

➤ Importancia de la tasa de aprendizaje

La tasa de aprendizaje es un hiperparámetro fundamental en el descenso por gradiente. Su valor determina la magnitud de los ajustes en cada iteración:

- Un η demasiado pequeño hace que el entrenamiento sea lento y puede quedar atrapado en mínimos locales.
- Un η demasiado grande puede hacer que el modelo oscile y no converja.
- Un η adecuado permite una convergencia eficiente hacia el mínimo global.



Descenso por gradiente por lotes (mini-batch)

El descenso por gradiente en mini batch es una técnica de optimización que busca un equilibrio entre la precisión del descenso por gradiente completo y la eficiencia computacional del descenso por gradiente estocástico. En lugar de utilizar todo el conjunto de datos o un solo ejemplo por iteración, divide los datos en pequeños lotes (mini batches) y actualiza los parámetros del modelo en función del gradiente calculado sobre cada lote.

➤ **Funcionamiento:**

1. División de los datos: Se divide el conjunto de entrenamiento en pequeños lotes de tamaño m .
2. Cálculo del gradiente: Se evalúa la función de costo y su gradiente sobre cada mini batch.
3. Actualización de los parámetros: Se ajustan los pesos y bias utilizando la regla:

$$w = w - \eta \frac{dj}{dw}_{\text{batch}}$$

donde $\frac{dj}{dw}_{\text{batch}}$ es la función de costo calculada sobre el mini batch.

4. Iteraciones: Se repite el proceso con cada mini batch hasta que el modelo converge.

➤ **Ventajas del Mini Batch:**

- Mejor eficiencia computacional en comparación con el descenso por gradiente tradicional, ya que permite aprovechar la paralelización en GPUs.
- Menos variabilidad que el descenso por gradiente estocástico, lo que ayuda a una convergencia más estable.
- Compromiso entre velocidad y precisión, ya que no necesita recorrer todo el dataset en cada actualización, pero aún captura suficiente información para realizar mejoras significativas en los parámetros.

Este método es ampliamente utilizado en la práctica debido a su balance entre rendimiento y estabilidad, especialmente cuando se trabaja con grandes volúmenes de datos.

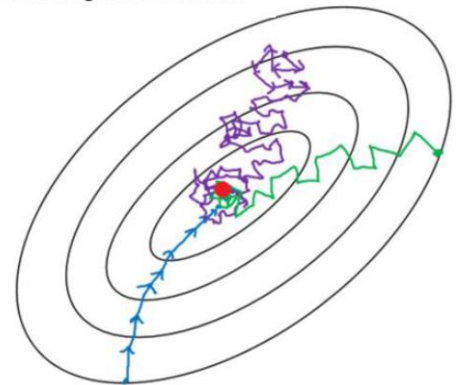
Descenso por Gradiente Estocástico (SGD)

El descenso por gradiente estocástico (SGD, por sus siglas en inglés) es una variante del descenso por gradiente que actualiza los parámetros del modelo después de evaluar el gradiente en un solo ejemplo de entrenamiento a la vez, en lugar de calcularlo sobre todo el conjunto de datos o un mini batch.

➤ Funcionamiento

1. Se selecciona un ejemplo aleatorio del conjunto de datos.
2. Se calcula el gradiente de la función de costo respecto a los parámetros del modelo usando solo ese ejemplo.
3. Se actualizan los parámetros
4. Se repiten estos pasos para varios ejemplos hasta alcanzar la convergencia.

— Batch gradient descent
— Mini-batch gradient Descent
— Stochastic gradient descent



Ventajas del SGD

- Mayor velocidad de actualización: al procesar un solo dato por iteración, los parámetros se actualizan más frecuentemente.
- Mejor exploración de la función de costo: debido a la aleatoriedad, puede escapar más fácilmente de mínimos locales en comparación con otros métodos.
- Adecuado para grandes volúmenes de datos: es útil cuando el conjunto de datos es muy grande y no cabe en memoria.

Desventajas del SGD

- Alta variabilidad en las actualizaciones: la actualización con un solo ejemplo introduce ruido, lo que puede dificultar la convergencia.
- Posibles oscilaciones alrededor del mínimo: debido a la naturaleza estocástica, el descenso puede ser menos estable que en otros métodos.

Otros algoritmos son:

Descenso de Gradiente con Momento

El descenso de gradiente con momento introduce la idea de acumular información de actualizaciones pasadas. En lugar de realizar ajustes solo con base en el gradiente actual, también incorpora un factor de inercia, lo que suaviza la trayectoria del descenso y acelera la convergencia.

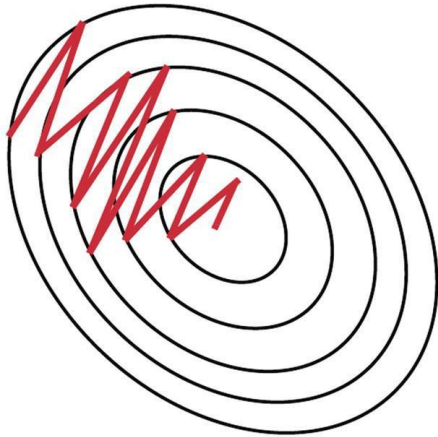
$$v(t) = \beta * v(t - 1) + (1 - \beta) * \delta(t)$$

Donde

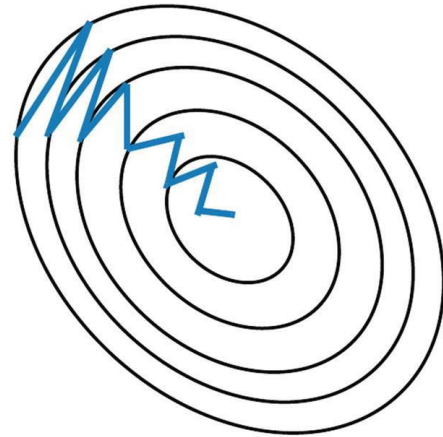
v(t): Es el valor actualizado del momento en el tiempo t.

$\beta \cdot v(t-1)$: Es la contribución del momento acumulado de la iteración anterior. El parámetro β (normalmente entre 0.9 y 0.99) controla cuánto del momento pasado se mantiene.

$(1-\beta) \cdot \delta(t)$: Es la contribución de la nueva información ($\delta(t)$), el gradiente de la función de costo.



Stochastic Gradient
Descent **without**
Momentum



Stochastic Gradient
Descent **with**
Momentum

Propagación de Raíz Cuadrática Media (RMSprop)

RMSprop es un método de optimización que ajusta dinámicamente la tasa de aprendizaje. Mantiene un promedio exponencialmente decreciente de los cuadrados de los gradientes para normalizar la actualización de los pesos. Este método ayuda a mitigar la oscilación del gradiente.

Explicación de las fórmulas:

- Actualización de las acumulaciones de los gradientes al cuadrado

$$v_{dw} = \beta \cdot v_{dw} + (1 - \beta) \cdot dw^2$$

$$v_{db} = \beta \cdot v_{dw} + (1 - \beta) \cdot db^2$$

- v_{dw} y v_{db} son acumulaciones exponenciales de los cuadrados de los gradientes.
- β (típicamente 0.9) controla la suavización exponencial: valores cercanos a 1 hacen que los valores pasados tengan más peso.
- Dw^2 y db^2 son los gradientes al cuadrado.

- Actualización de los pesos y bias

$$W = W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}} + \epsilon}$$

$$b = b - \alpha \cdot \frac{db}{\sqrt{v_{db}} + \epsilon}$$

- α es la tasa de aprendizaje.
- ϵ es un pequeño valor para evitar divisiones por cero.
- Normalizando el gradiente con la raíz cuadrada de v_{dw} o v_{db} , evitamos que los valores grandes hagan que los pesos cambien demasiado rápido.

Estimación Adaptativa del Momento (Adam)

Adam combina las ideas de momento y RMSprop para mejorar la estabilidad del entrenamiento. Mantiene dos estimaciones en paralelo: el promedio móvil de los gradientes y el promedio móvil de los cuadrados de los gradientes.

Momento:

$$V_{dw} = \beta_1 * V_{dw} + (1 - \beta_1) * dw$$

$$V_{db} = \beta_1 * V_{db} + (1 - \beta_1) * db$$

Rmsprop:

$$S_{dw} = \beta_2 * S_{dw} + (1 - \beta_2) * dw^2$$

$$S_{db} = \beta_2 * S_{db} + (1 - \beta_2) * db^2$$

Actualización de hiperparámetro

$$W = W - \alpha \frac{V_{dw}}{\sqrt{S_{dw} + 10^{-9}}}$$

$$b = b - \alpha \frac{V_{db}}{\sqrt{S_{db} + 10^{-9}}}$$

En resumen, las funciones de optimización son esenciales en el entrenamiento de redes neuronales artificiales, ya que permiten ajustar los pesos del modelo para minimizar el error. Métodos como el descenso por gradiente, en sus variantes estocástica y por mini-batch, ofrecen distintos compromisos entre precisión y eficiencia. Algoritmos más avanzados, como RMSprop y

Adam, optimizan la estabilidad y velocidad de convergencia mediante la adaptación dinámica de la tasa de aprendizaje. La elección del optimizador adecuado depende en gran medida del tamaño de la base de datos: en conjuntos pequeños, métodos como el descenso por gradiente estocástico suelen ser suficientes, mientras que, en bases de datos masivas, optimizadores más sofisticados permiten una convergencia más eficiente y estable.

Selección de Hiperparámetros

¿Qué es un hiperparámetro?

En el contexto del aprendizaje automático y específicamente en las redes neuronales artificiales (RNA), los hiperparámetros son valores que determinan el comportamiento del proceso de entrenamiento, pero no son aprendidos por el modelo durante dicho entrenamiento. Es decir, mientras los pesos y sesgos se ajustan automáticamente mediante algoritmos de optimización, los hiperparámetros deben ser definidos previamente por el usuario.

Algunos ejemplos típicos de hiperparámetros incluyen:

- La **tasa de aprendizaje** (α)
- El **número de épocas**
- El **tamaño del lote** (batch size)
- El **número de capas ocultas y de neuronas por capa**
- La **función de activación**
- El **algoritmo de optimización**

En esta sección nos enfocaremos particularmente en los hiperparámetros que más influyen en la arquitectura del modelo: la cantidad de **capas ocultas** y la **cantidad de neuronas** por capa, sin perder de vista cómo estas decisiones interactúan con fenómenos como el **sobreajuste** (overfitting) y el **subajuste** (underfitting).

Número de capas ocultas

Determinar cuántas capas ocultas debe tener una RNA no es una ciencia exacta. Existen recomendaciones empíricas, algunas fundamentadas teóricamente y otras simplemente prácticas, pero ninguna garantiza resultados óptimos sin experimentación. Entre los enfoques más conocidos se encuentran:

- Según **Lippmann (1987)**, una sola capa oculta puede ser suficiente para resolver problemas complejos, siempre que contenga una cantidad adecuada de neuronas (al menos tres veces el número de entradas).

- Si los **datos son linealmente separables**, no se requiere ninguna capa oculta. Un perceptrón simple puede ser suficiente.
- Para **problemas de baja complejidad** o con pocos atributos, una o dos capas ocultas suelen ser eficaces.
- Para problemas más complejos, se pueden utilizar entre **tres y cinco capas ocultas**, aunque esto incrementa la complejidad computacional y el riesgo de sobreajuste.

Otra estrategia útil es aplicar **procedimientos de poda**, que consisten en:

1. Iniciar con una red grande y eliminar progresivamente unidades y conexiones hasta alcanzar un modelo eficiente.
2. Comenzar con una red pequeña e ir incrementando tamaño según se detecten mejoras en desempeño.

Número de neuronas por capa

Determinar la cantidad de neuronas adecuadas para cada capa es igualmente delicado. Existen varias **reglas heurísticas** que orientan esta selección:

- El número de neuronas en las capas ocultas debe situarse entre el tamaño de la **capa de entrada** y el de la **capa de salida**.
- Puede aplicarse la regla de los **2/3**: el número de neuronas ocultas debe ser dos tercios del número de entradas más el número de salidas.
- Otra recomendación es no superar el **doble** del número de entradas.

Desde una perspectiva teórica, autores como **Hecht-Nielsen (1990)** y **Lippmann (1987)**, basándose en el teorema de Kolmogórov, sostienen que una sola capa oculta con **$(2N + 1)$** neuronas (donde N es el número de variables de entrada) puede aproximar cualquier función continua.

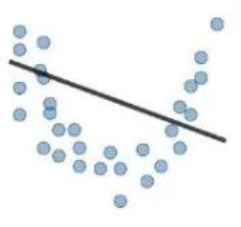

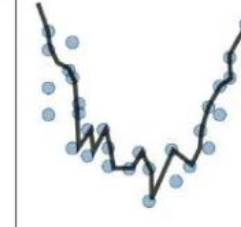
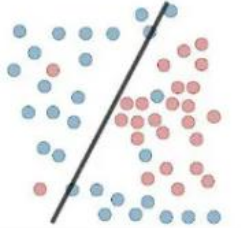
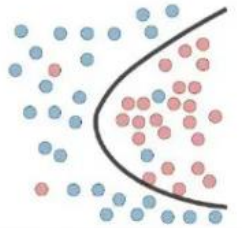
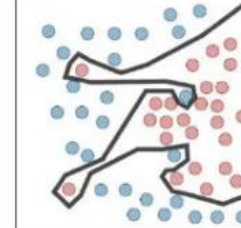


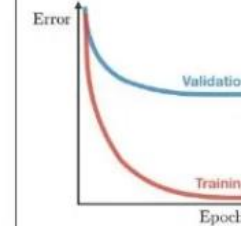
Algunas **reglas empíricas adicionales** incluyen:

- **Regla de la pirámide geométrica**: se sugiere que el número de neuronas disminuya progresivamente capa a capa desde la entrada hacia la salida.
- **Regla 2z1**: el número de neuronas ocultas no debe superar el doble del número de variables de entrada.

En la práctica, suele ser útil probar distintas configuraciones mediante validación cruzada o conjuntos de validación independientes, hasta encontrar una arquitectura con buen rendimiento general sin caer en sobreajuste.

Sobreajuste y subajuste

- El **sobreajuste (overfitting)** ocurre cuando el modelo aprende patrones específicos del conjunto de entrenamiento, incluyendo ruido y valores atípicos, y pierde capacidad de generalización. Se manifiesta cuando el rendimiento en entrenamiento es muy alto, pero muy bajo en validación o prueba.
- El **subajuste (underfitting)**, en cambio, se da cuando el modelo no es capaz de capturar la complejidad del patrón subyacente en los datos. En este caso, el rendimiento es pobre tanto en entrenamiento como en validación.

	Underfitting	Just right	Overfitting
Symptoms	<ul style="list-style-type: none"> - High training error - Training error close to test error - High bias 	<ul style="list-style-type: none"> - Training error slightly lower than test error 	<ul style="list-style-type: none"> - Low training error - Training error much lower than test error - High variance
Regression			
Classification			
Deep learning			
Remedies	<ul style="list-style-type: none"> - Complexify model - Add more features - Train longer 		<ul style="list-style-type: none"> - Regularize - Get more data

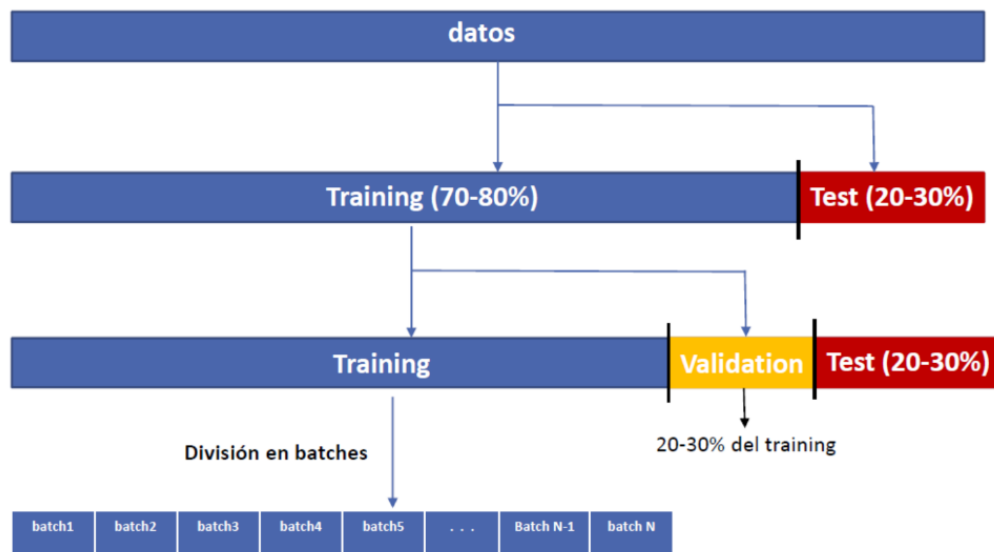
La arquitectura del modelo, es decir, la cantidad de capas y neuronas está directamente relacionada con este balance. Una red muy pequeña tenderá al subajuste; una red excesivamente compleja, al sobreajuste. Aquí es donde la experimentación con diferentes configuraciones de hiperparámetros se vuelve fundamental.

División del conjunto de datos: Entrenamiento, Validación y Prueba

La separación del conjunto de datos es esencial para medir la capacidad de generalización del modelo y evitar el autoengaño estadístico.

La práctica habitual consiste en dividir el dataset en tres subconjuntos:

- **Entrenamiento** (training set): entre el 60% y 90% del total, se utiliza para ajustar los pesos del modelo.
- **Validación** (validation set): entre el 5% y 20%, se usa para evaluar configuraciones de hiperparámetros y prevenir el sobreajuste.



- **Prueba** (test set): entre el 5% y 20%, se reserva para una evaluación final, simulando cómo se comportaría el modelo ante datos completamente nuevos.

Es importante destacar que los datos deben estar bien balanceados en cada subconjunto para evitar sesgos. En problemas con clases desbalanceadas, puede ser necesario aplicar técnicas de sobremuestreo o submuestreo.

Comparación de RNAs con estimación geoestadística

Implementación

Implementación en python: definición de unidades geológicas

Implementación en python: Incorporación de ley al modelo

Visualización con SGeMS