Trabajo de Teoria de TID: Redes Neuronales

Componentes.

- Arturo Cortés Sánchez
- Abel José Sánchez Alba

Índice:

Historia de las Redes Neuronales:

A finales de la década de 1940, Donald O. Hebb creó una hipótesis de aprendizaje basada en el mecanismo de plasticidad neuronal que se conoce como aprendizaje hebbiano. El aprendizaje hebbiano es un aprendizaje no supervisado. Esto evolucionó en modelos de potenciación a largo plazo. Los investigadores comenzaron a aplicar estas ideas a los modelos computacionales en 1948 con las máquinas de tipo B de Turing. En 1954 Farley y Clark utilizaron por primera vez máquinas computacionales, entonces llamadas "calculadoras", para simular una red hebbiana. En 1956 Rochester, Holland, Habit y Duda, crearon otras máquinas de computación de redes neuronales. En 1958 Rosenblatt creó el perceptrón, un algoritmo para el reconocimiento de patrones. Con notación matemática, Rosenblatt describió circuitos que no estaban en el perceptrón básico, como el exclusivo-o circuito que no podía ser procesado por las redes neuronales de la época. En 1959, un modelo biológico propuesto por los premios Nobel Hubel y Wiesel se basó en su descubrimiento de dos tipos de células en la corteza visual primaria: células simples y células complejas. Las primeras redes funcionales con muchas capas fueron publicadas por Ivakhnenko y Lapa en 1965.

La investigación se estancó después de la investigación de Minsky y Papert, quienes descubrieron dos problemas clave con redes neuronales. La primera era que los perceptrones básicos eran incapaces de procesar la función XOR. La segunda era que las computadoras no tenían suficiente poder de procesamiento para manejar efectivamente el trabajo requerido por redes neuronales de gran tamaño. El progreso en el desarrollo de las redes neuronales se ralentizó hasta que los ordenadores alcanzaron una potencia de procesamiento mucho mayor.

En 1975 Paul Werbos inventa el algoritmo de backpropagation o retropropagación, lo que renovó el interés por las redes neuronales. Este algoritmo permite el entrenamiento práctico de las redes multicapa. La retropropagación distribuye el error de vuelta a través de las capas, modificando los pesos en cada nodo.

Las máquinas de vectores de apoyo y otros métodos más simples, como los clasificadores lineales, superaron gradualmente las redes neuronales. Sin embargo, las redes neuronales transformaron dominios como la predicción de las estructuras de las proteínas.

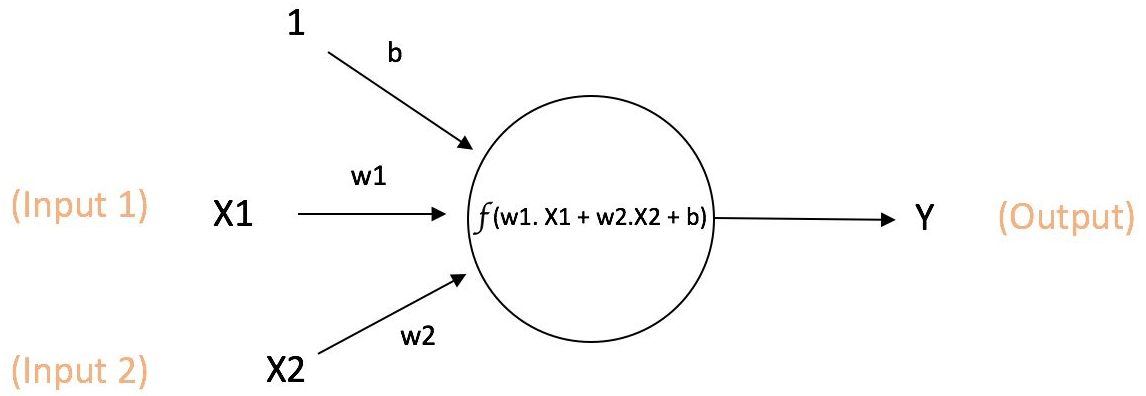
En 1992, se introdujo el sistema "max-pooling" para ayudar ayudar en el reconocimiento de objetos 3D. En 2010, el entrenamiento de la retropropagación a través del "max-pooling" fue acelerado por GPUs y se demostró que funcionaba mejor que otras variantes de "pooling".

Perceptron

Frank Rosenblatt, un psicólogo americano, propuso el modelo clásico de perceptrón en 1958. Su idea fue más refinada y cuidadosamente analizada por Minsky y Papert en 1969.

El modelo de perceptrón es un modelo computacional más general que la neurona de McCulloch-Pitts. Toma una entrada, la agrega (suma ponderada) y devuelve 1 sólo si la suma agregada es mayor que cierto umbral, de lo contrario devuelve 0. Un erceptrón sólo puede ser usado para implementar funciones linealmente separables. Toma tanto las entradas reales como las booleanas y les asocia un conjunto de pesos, junto con un sesgo.

La unidad básica de cálculo en una red neuronal es la neurona, a menudo llamada nodo o unidad. Recibe la entrada de algunos otros nodos, o de una fuente externa y calcula una salida. Cada entrada tiene un peso asociado (w), que se asigna en función de su importancia relativa con respecto a las demás entradas. El nodo aplica una función f (definida más adelante) a la suma ponderada de sus entradas.



$$\text{Output of neuron} = Y = f(w1.\,X1 + w2.X2 + b)$$

La red anterior toma las entradas numéricas X1 y X2 y tiene los pesos w1 y w2 asociados a esas entradas. Además, hay otra entrada 1 con el peso b (llamado el Sesgo) asociado a ella.
La salida Y de la neurona se calcula como se muestra en la imagen. La función f es no lineal y se llama la Función de Activación. El propósito de la función de activación

es introducir la no linealidad en la salida de una neurona. Esto es importante porque la mayoría de los datos del mundo real son no lineales y queremos que las neuronas aprendan estas representaciones no lineales.

Cada función de activación toma un solo número y realiza una cierta operación matemática fija en él. Hay varias funciones de activación que puede encontrar en la práctica:

- Sigmoide: toma una entrada de valor real y la escala en un rango entre 0 y 1

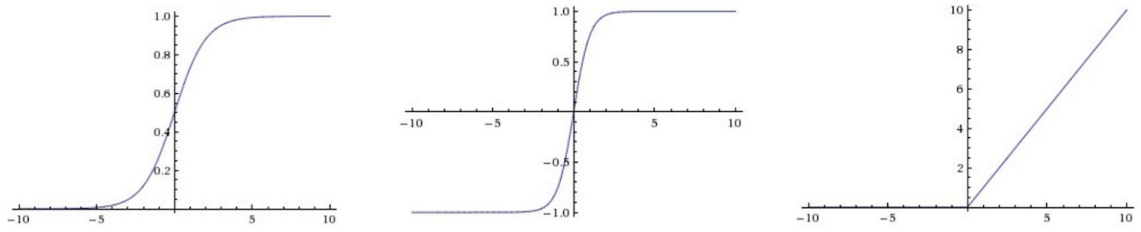
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

- tanh: toma una entrada de valor real y la escala en el rango [-1, 1]

$$f(x) = tanh(x)$$

- ReLU: ReLU significa unidad lineal rectificada. Toma una entrada de valor real y si es negativa la reemplaza con cero.

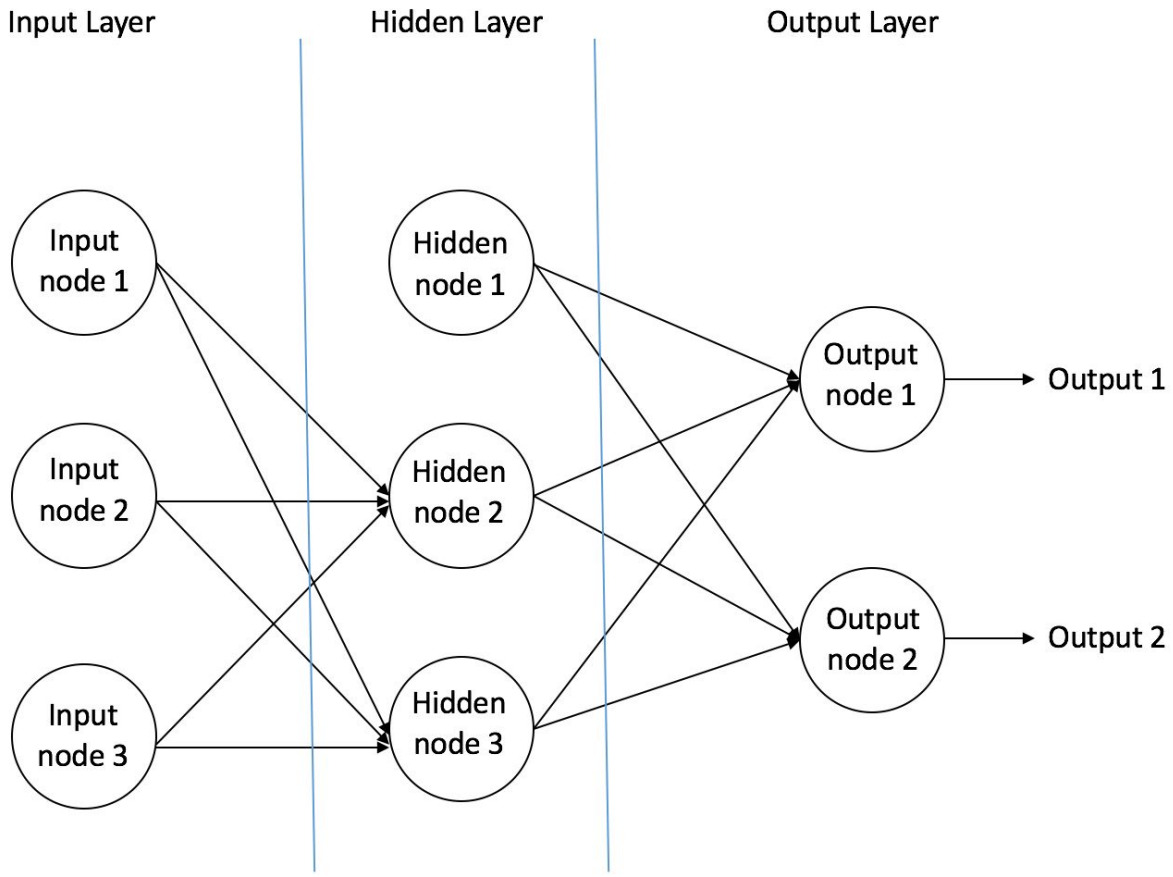
$$f(x) = max(0, x)$$



Sigmoid tanh ReLU

Estructura de una red neuronal

La red neuronal prealimentada fue la primera red neuronal artificial ideada. Contiene múltiples neuronas (nodos) dispuestas en capas. Los nodos de las capas adyacentes tienen conexiones entre ellos. Todas estas conexiones tienen pesos asociados a ellas.



Una red neuronal prealimentada consiste en tres tipos de nodos:

- Nodos de entrada - Los nodos de entrada proporcionan información del mundo exterior a la red y juntos se denominan "Capa de entrada". No se realiza ningún cálculo en ninguno de los nodos de entrada - sólo pasan la información a los nodos ocultos.

- Nodos ocultos - Los nodos ocultos no tienen una conexión directa con el mundo exterior (de ahí el nombre "oculto"). Realizan cálculos y transfieren información de los nodos de entrada a los nodos de salida. Un conjunto de nodos ocultos forma una "Capa oculta". Mientras que una red prealimentada

sólo tendrá una capa de entrada y una capa de salida, puede tener de cero a varias capas ocultas.
- Nodos de salida - Los nodos de salida se denominan colectivamente "Capa de salida" y son responsables de los cálculos y de la transferencia de información de la red al mundo exterior.

En una red prealimentada, la información se mueve en una sola dirección - avance - desde los nodos de entrada, a través de los nodos ocultos (si los hay) y a los nodos de salida. No hay ciclos o bucles en la red. Esta propiedad de las redes prealimentadas es diferente de las redes neuronales recurrentes en las que las conexiones entre los nodos forman un ciclo.

Función de error

Una función de coste o función de error es una función que asigna un evento o los valores de una o más variables a un número real que representa intuitivamente algún "coste" asociado con el evento. Un problema de optimización busca minimizar una función de error.

En estadística, normalmente se utiliza una función de coste para la estimación de parámetros, y el evento en cuestión es una función de la diferencia entre los valores estimados y los verdaderos para una instancia de datos. El concepto, fue reintroducido en la estadística por Abraham Wald a mediados del siglo XX.
En el contexto de la economía, por ejemplo, suele tratarse de un coste económico o de un arrepentimiento. En la clasificación, es la penalización por una clasificación incorrecta de un ejemplo.

En el aprendizaje automático y la optimización matemática, las funciones de error para la clasificación son funciones de error computacionalmente factibles que representan el precio pagado por la inexactitud de las predicciones en los problemas de clasificación. Dado el espacio de todos los posibles inputs X y un con salidas Y={0,1}. El objetivo de la clasificación es encontrar una función que, dada una entrada, prediga de forma exacta la salida. Por otro lado, el objetivo del aprendizaje es reducir esa función de error o de coste.

Estas funciones normalmente vienen definidas como combinaciones de los valores predichos por el algoritmo y los valores de salida reales del problema. Existen muchos tipos de funciones de error, aunque normalmente se trabaja con las siguientes que dependen del problema.

1. Problema de regresión

Un problema en el que se predice una cantidad de valor real. En la capa de salida se tiene un nodo con una unidad de activación lineal, aunque se puede utilizar una activación sigmoide si la regresión no es lineal. Como función de error se emplea el Error cuadrático medio (MSE).

2. Problema de clasificación binaria

Un problema en el que se clasifica un ejemplo como perteneciente a una de dos clases. El problema, en esencia, es la predicción de la probabilidad de que un ejemplo pertenezca a la clase uno, por ejemplo, la clase a la que se asigna el valor entero 1, mientras que a la otra clase se le asigna el valor 0. En la capa de salida se tiene un nodo con una unidad de activación sigmoide.Como función de error se emplea la entropía cruzada (Cross-Entropy), también conocida como error logarítmico.

3. Problema de clasificación de clases múltiples

Un problema en el que se clasifica un ejemplo como perteneciente a una de más de dos clases. El problema consta como la predicción de la probabilidad de que un ejemplo pertenezca a cada clase. En la capa de salida se tiene un nodo para cada clase usando la función de activación de softmax. Como función de error se emplea la entropía cruzada (Cross-Entropy), también conocida como error logarítmico.

Backpropagation

En el aprendizaje automático, la propagación hacia atrás es un algoritmo ampliamente utilizado para el entrenamiento de redes neuronales. Existen generalizaciones de propagación hacia atrás para otras redes neuronales artificiales, y para funciones en general. Al entrenar una red neuronal, la propagación hacia atrás calcula el gradiente de la función de error con respecto a los pesos de la red para un único ejemplo de entrada-salida, y lo hace de forma eficiente, a diferencia de un cálculo directo del gradiente con respecto a cada peso individualmente. Esta eficiencia hace factible el uso de métodos de gradiente para la formación de redes multicapa, actualizando los pesos para minimizar el error. El descenso de gradiente, o variantes como el descenso de gradiente estocástico, son los algoritmos más utilizados. El algoritmo de propagación hacia atrás funciona calculando el gradiente de la función de error con respecto a cada peso por la regla de la cadena, calculando el gradiente de capa en capa, iterando hacia atrás desde la última capa para evitar cálculos redundantes de términos intermedios en la regla de la cadena.

Para comprender la derivación matemática del algoritmo de backpropagation, es necesario conocer la relación entre la salida real de una neurona y la salida correcta para un ejemplo de entrenamiento particular. Consideremos una red neuronal

simple con dos unidades de entrada, una unidad de salida y ninguna unidad oculta, y en la que cada neurona utiliza una salida lineal.

Inicialmente, antes de entrenar, los pesos se fijarán al azar. Luego la neurona aprende de los ejemplos de entrenamiento, que en este caso consisten en un conjunto de tuplas (x1,x2,t) donde x1 y x2 son las entradas de la red y t es la salida correcta (la salida que la red debe producir dadas esas entradas, cuando ha sido entrenada). La red inicial, dados x1 y x2, calculará una salida "y" que probablemente difiera de t (dados los pesos aleatorios). Una función de error, L(t,y) se utiliza, como se ha indicado antes, para medir la discrepancia entre la salida objetivo t y la salida calculada y.

Consideremos la red en un solo caso de entrenamiento: (1, 1, 0). Así, la entrada x1 y x2 son 1 y 1 respectivamente y la salida correcta, t es 0. Ahora, si se traza la relación entre la salida de la red y en el eje horizontal y el error E en el eje vertical, el resultado es una parábola. El mínimo de la parábola corresponde a la salida y que minimiza el error E. En un entrenamiento de un solo caso, el mínimo también toca el eje horizontal, lo que significa que el error será cero y la red puede producir una salida y que coincide exactamente con la salida objetivo t.

Sin embargo, la salida de una neurona depende de la suma ponderada de todas sus entradas:

$$y = x1w1 + x2w2$$

Donde w1 y w2 son los pesos en la conexión de las unidades de entrada a la unidad de salida. Por lo tanto, el error también depende de los pesos de entrada a la neurona, que es lo que en última instancia debe ser cambiado en la red para permitir el aprendizaje. Si cada peso se traza en un eje horizontal separado y el error en el eje vertical, el resultado es parabólico. Para una neurona con K pesos, la misma gráfica requeriría un paraboloide elíptico de dimensiones K+1.

Un algoritmo comúnmente utilizado para encontrar el conjunto de pesos que minimiza el error es el descenso de gradiente. La propagación hacia atrás se utiliza entonces para calcular la dirección de descenso más pronunciada de una manera eficiente.

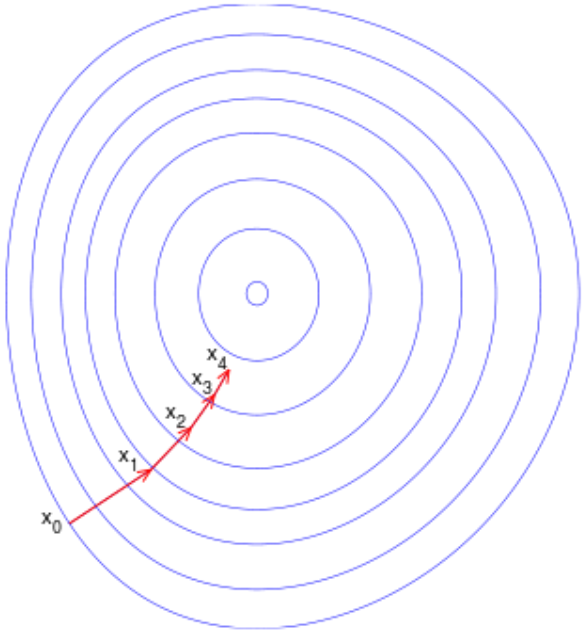
Descenso del gradiente.

El descenso de gradiente es un algoritmo de optimización iterativo de primer orden para encontrar un mínimo local de una función diferenciable. La idea consiste en dar pasos repetidos en la dirección opuesta al gradiente (o gradiente aproximado) de la

función en el punto actual, porque es la dirección de descenso más pronunciada. A la inversa, si se dan pasos en la dirección del gradiente se obtendrá un máximo local de esa función; el procedimiento se conoce entonces como ascenso de gradiente.

El descenso del gradiente se basa en la observación de que si la función multivariable F(x) definida se puede diferenciar en un entorno de un punto a, entonces F disminuye más rápido si uno avanza desde el punto a en la dirección contraria al gradiente de F.

Se puede garantizar la convergencia a un mínimo local. Cuando la función F es convexa, todos los mínimos locales son también mínimos globales, por lo que en este caso el descenso de gradiente puede converger hacia la solución global.



Este proceso se ilustra en la imagen superior. Aquí se supone que F está definida en el plano, y que su gráfico tiene forma de cuenco. Las curvas azules son las líneas de contorno, es decir, las regiones en las que el valor de F es constante. La

flecha roja que se origina en un punto muestra la dirección del gradiente negativo en ese punto. Véase que el gradiente (negativo) en un punto es ortogonal a la línea de contorno que pasa por ese punto. Vemos que el descenso del gradiente nos lleva al fondo del cuenco, es decir, al punto en el que el valor de la función F es mínimo.

Código de la demostración:

A continuación se tiene un código escrito en python que ilustra el funcionamiento de una red neuronal y cómo evoluciona su aprendizaje.

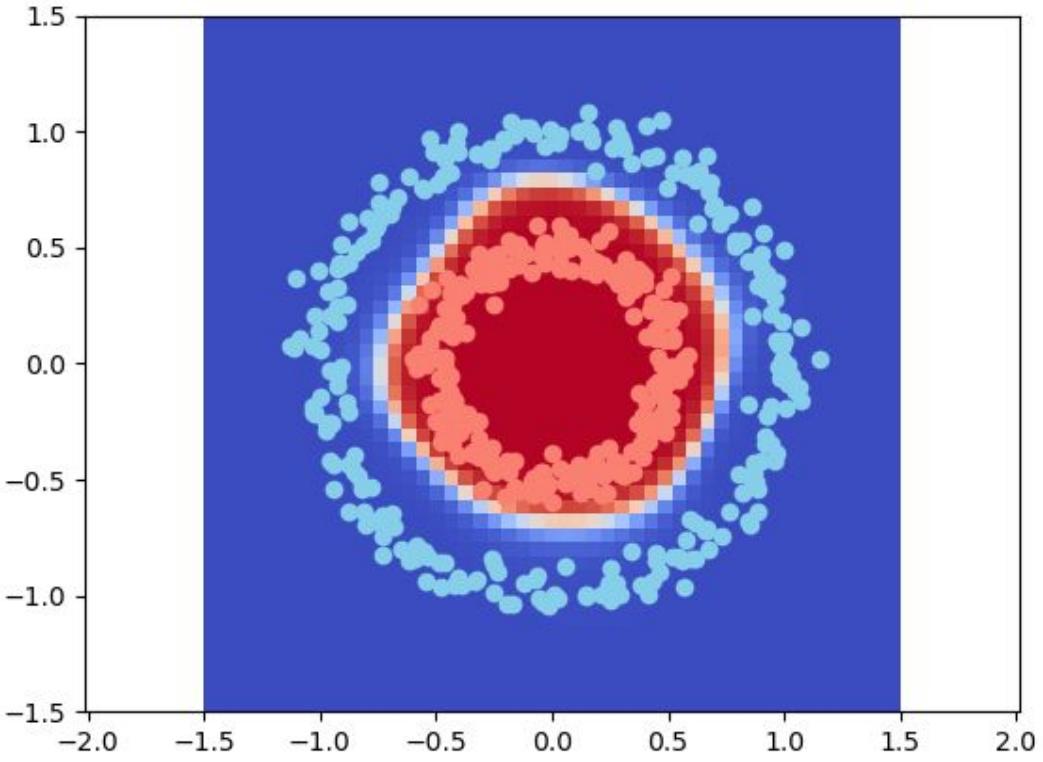
```
import numpy as np
import scipy as sc
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

n = 500
p = 2
X, Y = make_circles(n_samples=n, factor=0.5, noise=0.05)
Y = Y[:, np.newaxis]


class capa_neuronas():

    def __init__(self, n_con, n_neur, act_f):
        self.act_f = act_f
        self.b = np.random.rand(1, n_neur)*2 - 1  # en [-1,1]
        self.W = np.random.rand(n_con, n_neur)*2 - 1  # en [-1,1]


# FUNCIONES DE ACTIVACIÓN
sigm = (lambda x: 1/(1+np.e**(-x)),
        lambda x: x*(1-x))  # sigmoide y derivada


def relu(x): return np.maximum(0, x)


def crear_red(topologia, act_f):
    red = []
    for l, capa in enumerate(topologia[:-1]):
        red.append(capa_neuronas(topologia[l], topologia[l+1], act_f))
    return red
```

```
topologia = [p, 4, 8, 4, 1]
red = crear_red(topologia, sigm)

l2_cost = (lambda Yp, Yr: np.mean((Yp-Yr)**2),
           lambda Yp, Yr: (Yp-Yr)
           )


def train(red_neuronal, X, Y, l2_cost, lr=0.1, train=True):

    out = [(None, X)]
    # paso hacia adelante
    for l, layer in enumerate(red_neuronal):
        z = out[-1][1] @ red_neuronal[l].W + red_neuronal[l].b
        a = red_neuronal[l].act_f[0](z)
        out.append((z, a))
    if train:
        # paso hacia atrás
        deltas = []
        for l in reversed(range(0, len(red_neuronal))):
            z = out[l+1][0]
            a = out[l+1][1]
            if l == len(red_neuronal)-1:
                deltas.insert(0, l2_cost[1](a,
Y)*red_neuronal[l].act_f[1](a))
            else:
                deltas.insert(0, deltas[0] @ _W.T *
                              red_neuronal[l].act_f[1](a))
            _W = red_neuronal[l].W


# DESCENSO DEL GRADIENTE
            red_neuronal[l].b = red_neuronal[l].b - \
                np.mean(deltas[0], axis=0, keepdims=True)*lr
            red_neuronal[l].W = red_neuronal[l].W - \
                out[l][1].T @ deltas[0] * lr

    return out[-1][1]


red_neuronal = crear_red(topologia, sigm)
loss = []
for i in range(2500):

    # entrenamos la red
    pY = train(red_neuronal, X, Y, l2_cost, 0.05)

    if i % 25 == 0:

        loss.append(l2_cost[0](pY, Y))
```

```
        res = 50

        _x0 = np.linspace(-1.5, 1.5, res)
        _x1 = np.linspace(-1.5, 1.5, res)

        _Y = np.zeros((res, res))

        for i0, x0 in enumerate(_x0):
            for i1, x1 in enumerate(_x1):
                _Y[i0, i1] = train(red_neuronal, np.array(
                    [[x0, x1]]), Y, l2_cost, 0.2, train=False)[0][0]


plt.pcolormesh(_x0, _x1, _Y, cmap="coolwarm")
plt.axis("equal")
plt.scatter(X[Y[:, 0] == 0, 0], X[Y[:, 0] == 0, 1], c="skyblue")
plt.scatter(X[Y[:, 0] == 1, 0], X[Y[:, 0] == 1, 1], c="salmon")
plt.show()
plt.plot(range(len(loss)), loss)
plt.show()
```

Resultado

Como podemos ver la red neuronal ha conseguido distinguir entre los dos conjuntos de datos:



Fuentes

https://en.wikipedia.org/wiki/Perceptron

https://en.wikipedia.org/wiki/History_of_artificial_neural_networks

https://en.wikipedia.org/wiki/Artificial_neural_network

https://web.csulb.edu/~cwallis/artificialn/History.htm

https://www.kdnuggets.com/2016/11/quick-introduction-neural-networks.html

https://www.youtube.com/watch?v=W8AeOXa_FqU

https://elvex.ugr.es/decsai/deep-learning/

http://neuralnetworksanddeeplearning.com/chap2.html

https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e

https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/

https://towardsdatascience.com/perceptron-the-artificial-neuron-4d8c70d5cc8d