PERCEPTRON

By: Jared Isaías Monje Flores - 217594478

6 DE FEBRERO DE 2024

CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS
UdeG

Modelo McCulloch-Pitts

El modelo de McCulloch-Pitts (También conocido como modelo de neurona artificial) es un modelo matemático que fue propuesto por Warren McCulloch y Walter Pitts en 1943. Es uno de los primeros modelos de la teoría de la inteligencia artificial y sigue siendo una parte importante de la investigación.

En el modelo de McCulloch-Pitts, una neurona se representa como un elemento básico que puede recibir señales de entrada y producir una señal de salida. Cada señal de entrada se asocia con un peso, que indica la importancia de la señal. La salida de la neurona se determina por una función de activación que evalúa la suma ponderada de las señales de entrada. Este modelo ha sido ampliamente utilizado como una base para el desarrollo de algoritmos de aprendizaje automático y para la investigación en la inteligencia artificial y la neurociencia computacional. Aunque es un modelo simplificado de una neurona real, ha sido muy útil para entender el funcionamiento de los sistemas de procesamiento de información en el cerebro.

El algoritmo del Perceptrón

El algoritmo del Perceptrón es un algoritmo de aprendizaje automático supervisado que se utiliza para clasificar datos en dos categorías. Se basa en el modelo de McCulloch-Pitts y es uno de los primeros algoritmos de aprendizaje automático desarrollados.

El algoritmo del Perceptrón funciona asignando pesos a las características de los datos de entrada y aplicando una función de activación para producir una salida. La función de activación es una función lineal que toma en cuenta la suma ponderada de las señales de entrada. Si la salida es mayor que cero, la entrada se clasifica en una categoría, y si la salida es menor o igual a cero, se clasifica en otra categoría. El algoritmo se ajusta iterativamente a los datos de entrenamiento, ajustando los pesos para minimizar el número de errores de clasificación. Este proceso de ajuste de pesos se llama “entrenamiento” y se realiza utilizando una técnica de optimización, como el gradiente descendente.

Aunque el algoritmo del Perceptrón es muy simple, ha sido ampliamente utilizado y ha demostrado ser efectivo en una amplia variedad de problemas de clasificación. Sin embargo, también tiene algunas limitaciones, como su capacidad limitada para manejar problemas de clasificación no lineales.

Por esta razón, se han desarrollado algoritmos más avanzados como los árboles de decisión, las redes neuronales y los algoritmos de aprendizaje profundo.

Parte 1 – Compuertas lógicas

Algoritmo del perceptrón

For e∈{1, 2 ,..., epochs}

For i∈{1 , 2,..., p }

$\hat{y}=\phi(w^T x+b)$, x ,$w\in\mathbb{R}^n$

$w\leftarrow w+\eta(y^{(i)}-\hat{y})x^{(i)}$

$b\leftarrow b+\eta(y^{(i)}-\hat{y})$

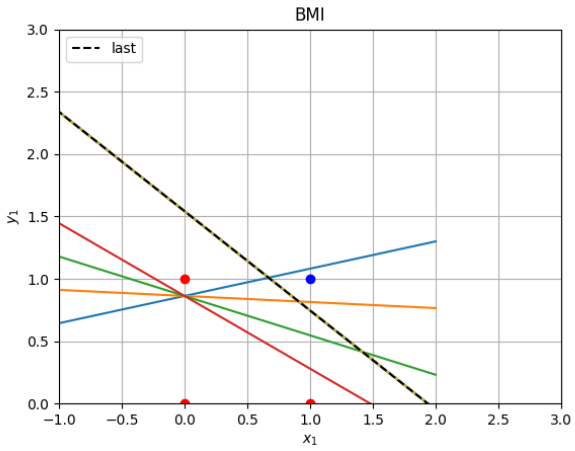

Figura 1: Compuerta lógica AND

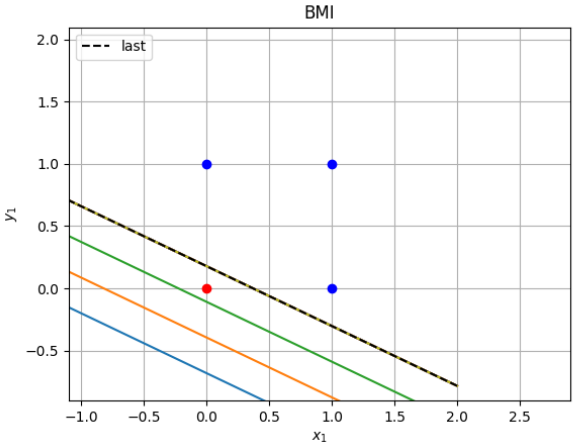

Figura 2: Compuerta lógica OR

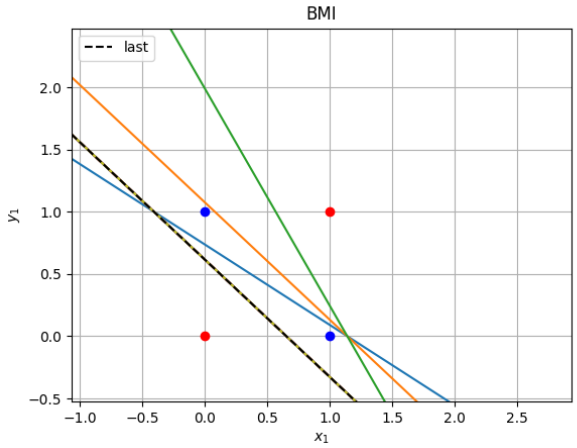

Figura 3: Compuerta lógica XOR

Parte 2 – índice de masa corporal

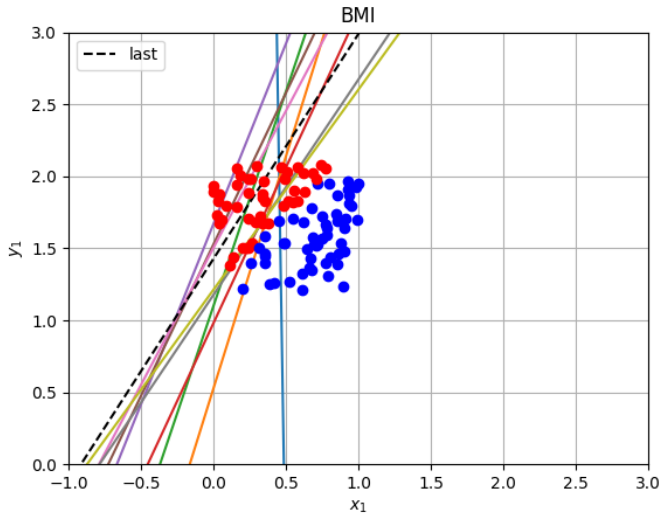

Figura 4: Índice de masa corporal - Datos de entrenamiento

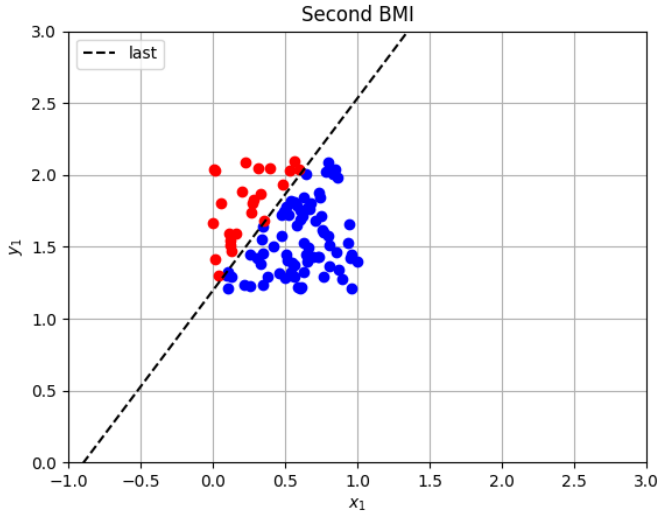

Figura 5: Índice de masa corporal – Datos nuevos, neurona entrenada

Conclusión

¿Por qué la compuerta XOR no puede ser aprendida por este modelo?

El modelo del perceptrón es una red neuronal de una sola capa que puede aprender a clasificar datos *linealmente seriales* (aquellos que se pueden separar con una línea recta), la compuerta XOR es un problema que no es linealmente separable, lo que significa que no se puede separar los puntos en dos clases usando una línea recta. En particular, los puntos de entrada (0,0) y (1,1) pertenecen a una clase, mientras que los puntos (0,1) y (1,0) pertenecen a la otra clase. Cuando se intenta entrenar un perceptrón para aprender la compuerta XOR, el modelo no es capaz de encontrar una línea recta que separe los puntos en dos clases.

¿Qué importancia tiene el factor de aprendizaje?

El factor de aprendizaje es un valor que se utiliza para ajustar la magnitud de los cambios en los pesos de la red durante el entrenamiento. El factor de aprendizaje controla qué tan rápido o lento la red neuronal ajusta sus pesos en función de los errores cometidos durante el entrenamiento.

La importancia del factor de aprendizaje radica en que puede afectar significativamente el rendimiento del modelo del perceptrón. Si el factor de aprendizaje es demasiado pequeño, el modelo puede tardar mucho tiempo en converger y, en algunos casos, puede quedarse atascado en un mínimo local. Si el factor de aprendizaje es demasiado grande, el modelo puede oscilar demasiado entre los pesos y no converger. Por lo tanto, elegir un factor de aprendizaje apropiado es crucial para lograr un buen rendimiento del modelo del perceptrón.

Código

```
#
# Perceptron
# By: Dexne
#
# Para tenemos mejores resultados, modificar los valores del
learning rate y las epocas
#

# Importamos matplotlib para graficar y numpy para calculos
import matplotlib.pyplot as plt
import numpy as np


# funcion para graficar los datos
def draw_2d(model, i, last = False) -> None:
    # variables a utilizar
    w1, w2, b = model.w[0], model.w[1], model.b
    li, ls = -2, 2 # limite inferior y sulperior

    if ( last ):
        plt.plot(
            [li, ls],
            [(1/w2)*(-w1*(li)-b), (1/w2)*(-w1*(ls)-b)],
            '--k', label='last'
        )

    else:
        plt.plot(
            [li, ls],
            [(1/w2)*(-w1*(li)-b), (1/w2)*(-w1*(ls)-b)]
        )
```

```
# Funcion para normalizar los datos
def get_normalized(N):
    Y = np.zeros(N)
    weight_lower, weight_upper = 35, 120 # Establecemos limites de
pesos
    height_lower, height_upper = 1.2, 2.1 # Y limite par las
alturas

    # Generamos los pesos y las alturas de manera aleatoria
    X = np.array([
        (weight_lower + (weight_upper - weight_lower) *
np.random.rand(N)),
        (height_lower + (height_upper - height_lower) *
np.random.rand(N))
    ])

    # Salida
    for i in range(N):
        if X[0, i] / X[1, i]**2 < 25:
            Y[i] = 0 # No tiene sobrepeso
        else:
            Y[i] = 1 # Tiene sobrepeso

    # Normalizacion de los datos
    nmin, nmax = min(X[0, :]), max(X[0, :])

    for i in range(N):
        X[0, i] = (( X[0, i] - nmin) / (nmax - nmin))

    return X, Y
```

```python
class Perceptron:
    # Constructor
    def __init__(self, n_input, learning_rate) -> None:
        # Inicializamos las variables
        self.w = -1 + 2 * np.random.rand(n_input)
        self.b = -1 + 2 * np.random.rand()
        self.eta = learning_rate

    # Funcion de prediccion
    def predict(self, X) -> bool:
        _, p = X.shape
        y_est = np.zeros(p)

        for i in range(p):
            y_est[i] = np.dot(self.w, X[:, i]) + self.b

            if ( y_est[i] >= 0 ):
                y_est[i] = 1
            else:
                y_est[i] = 0

        return y_est

    # funcion entrenamiento
    def fit(self, X, Y, epoch = 10) -> None:
        it = 1
        _, p = X.shape
        for _ in range(epoch):
            if ( _ == epoch-1 ):
                draw_2d(neuron, it, True)
            else:
                draw_2d(neuron, it)
```

```
            for i in range(p):
                y_est = self.predict(X[:, i].reshape(-1, 1))
                self.w += self.eta * (Y[i] - y_est) * X[:, i]
                self.b += self.eta * (Y[i] - y_est)

                it+=1

if __name__=='__main__':
    neuron = Perceptron(2, 0.2)
    problem = "gates"   #[ "gates" "bmi" ]

    # Compuertas logicas
    if problem == "bmi":
        X = np.array([
            [0, 0, 1, 1],
            [0, 1, 0, 1]
        ])

        #Y = np.array([0,0,0,1]) # AND
        #Y = np.array([0,1,1,1]) # OR
        #Y = np.array([0,1,1,0]) # XOR

    # BMI
    else:
        N = 100 # Generamos 100 pesos y alturas de manera
aleatoria
        X, Y = get_normalized(N) # normalizamos los datos


    print('W:\t', neuron.w )
    print('P:\t', neuron.predict(X))
```

```
    neuron.fit(X, Y)
    print('W:\t', neuron.w )
    print('P:\t', neuron.predict(X))

    # Dibujamos los datos
    _, p = X.shape
    for i in range(p):
        if(not Y[i]):
            plt.plot(X[0, i], X[1, i], 'or')
        else:
            plt.plot(X[0, i], X[1, i], 'ob')


    plt.title('BMI')
    plt.grid('on')
    plt.xlim([-1, 3])
    plt.ylim([0, 3])
    plt.xlabel(r'$x_1$')
    plt.ylabel(r'$y_1$')
    plt.legend(loc='upper left')
    plt.show()


    # Segunda fecha prueba del BMI
    if problem == "bmi":
        plt.figure()
        X, Y = get_normalized(N)

        for i in range(N):
            Y[i] = neuron.predict(X[:, i].reshape(-1, 1))

        # Dibujamos los puntos
```

```
    _, p = X.shape
    for i in range(p):
        if(not Y[i]):
            plt.plot(X[0, i], X[1, i], 'or')
        else:
            plt.plot(X[0, i], X[1, i], 'ob')

    draw_2d(neuron, 0, True)

    plt.title('Second BMI')
    plt.grid('on')
    plt.xlim([-1, 3])
    plt.ylim([0, 3])
    plt.xlabel(r'$x_1$')
    plt.ylabel(r'$y_1$')
    plt.legend(loc='upper left')
    plt.show()
```