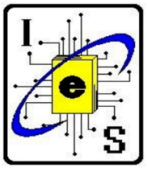


ACTIVIDAD DE TRABAJO COLABORATIVO

ANÁLISIS DEL PERCEPTRÓN SIMPLE

PRESENTADO POR:
CRISTIAN JAVIER ARENAS COGOLLO

DIRIGIDO A: DOCENTE MANUEL ALEJANDRO OSPINA ALARCÓN

INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE CARTAGENA

FACULTAD DE INGENIERÍA

PROGRAMA DE INGENIERÍA DE SISTEMAS

CARTAGENA DE INDIAS D.T Y C - BOLÍVAR COLOMBIA

2024

Tabla de contenido

1. Problema 1: La paloma que busca alimento

1.1 Descripción del problema

Supongamos que deseamos saber si una paloma es capaz de aprender la manera de conseguir la comida. Para ello, se colocan dos pulsadores, uno al lado del otro y cada pulsador puede estar encendido o apagado. Para conseguir la comida tiene que picar en cualquiera de los pulsadores, salvo cuando los dos están apagados que debe picar solamente en el pulsador derecho

Para ello deben representar el problema mediante un vector bipolar (x1,x2,x3), donde x1 toma el valor 1 ó –1 según que el pulsador de la izquierda esté encendido o no, respectivamente; x2 toma el valor 1 ó –1 según que el pulsador de la derecha esté encendido o no, y x3 toma el valor 1 ó –1 según que la paloma pique en el pulsador de la izquierda o en el de la derecha, respectivamente.

1.2 Propuesta de la solución

Primero se define la tabla que modela las distintas posibilidades del perceptrón, en este caso, el contexto que determina cuando la paloma conseguirá el alimento o no.

X1	X2	X3	COME
1	1	1	1
1	1	-1	1
1	-1	1	1
1	-1	-1	1
-1	1	1	1
-1	1	-1	1
-1	-1	1	1
-1	-1	-1	-1

Tabla 1: Combinaciones del problema

Fuente Elaboración Propia

Se puede concluir que en todas las situaciones la paloma podrá comer, excepto en una sola, donde los pulsadores están apagados y la paloma presiona el izquierdo, lo que le imposibilita conseguir alimento.

Por otro lado, mediante el lenguaje Python se aprovechó las librerías *numpy*, y *matplotlib*, para hacer más práctico y eficiente el algoritmo.

1.2.1 Implementación del Algoritmo

El algoritmo está compuesto por una clase para representar el perceptrón; y diferentes métodos para realizar las funciones de activación, el entrenamiento de la red, y la impresión de los valores por pantalla.

```
import random
import numpy as np
import matplotlib.pyplot as plt

class PerceptronPaloma:
    def __init__(self):
        self.pesos = [random.uniform(-1, 1) for _ in range(3)] #valores aleato
        self.bias = random.uniform(0, 1) # valores aleatorios para el peso del
        self.tasaAprendizaje = 0.1 # valor de la tasa de aprendizaje
```

Imagen 1: Definición del Perceptrón que modela a la Paloma

Fuente: Elaboración propia

Mediante el uso de la librería *numpy* y la librería *random*, se utilizan funciones para definir valores aleatorios a cada una de las variables de entrada (3 en total), y el bias; además, se define un valor constante para la tasa de aprendizaje, de 0.1, el cual ayudará a variar en menor grado la actualización de los pesos y bias.

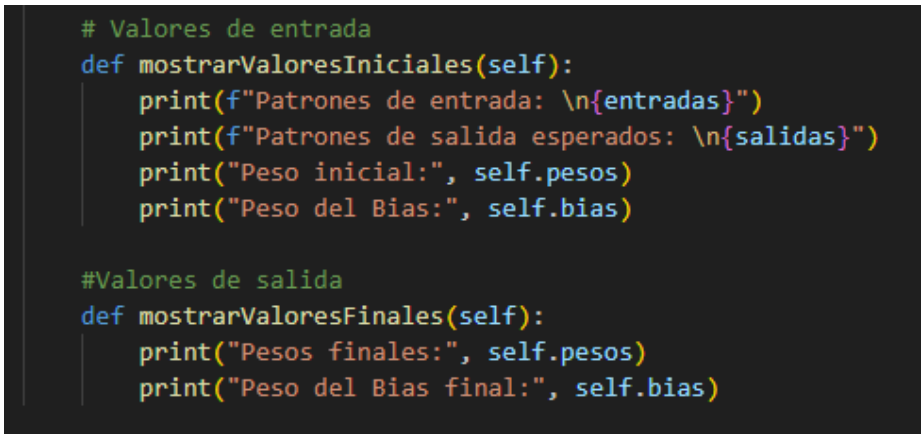
```python
# funcion de activacion
def funcionActivacion(self, suma):
    return 1 if suma > 0 else -1

# logica de entrenamiento del perceptron
def entrenar(self, inputs, desired_outputs):
    errores_epoca = []
    for iteraciones in range(100):  # Número de épocas de entrenamiento
        error = False
        error_total = 0
        for i in range(len(inputs)):
            suma = np.dot(inputs[i], self.pesos) + self.bias
            output = self.funcionActivacion(suma)
            error = error or (output != desired_outputs[i])
            error_term = desired_outputs[i] - output

            self.pesos += self.tasaAprendizaje * error_term * np.array(inputs[i])
            self.bias += self.tasaAprendizaje * error_term

            error_total +=abs(error_term)
        errores_epoca.append(error_total)

        if not error:
            break
    return errores_epoca
```

Imagen 2: Función de activación y de entrenamiento

Fuente: Elaboración propia

La función de entrenamiento se encargará analizar todas las posibilidades buscando que dependiendo las condiciones de entrada, esta pueda acertar con el resultado; para esto recibe por parámetro una lista de variables de entrada, y las salidas esperadas. Posteriormente inicializa una lista para guardar los errores encontrados y más adelante generar una gráfica a partir de ella; el entrenamiento se dará en un límite finito de 100 épocas, donde el perceptrón puede aprender en una iteración menor o igual a 100.

El entrenamiento se basa en calcular las variables de entrada con sus respectivos pesos, y el bias, para luego evaluar ese valor mediante la función de activación hardlim, donde si el valor es mayor a 0, toma un 1, o en caso contrario, un .1; en base a ese resultado se determina si para esa entrada que se está evaluando es necesario recalcular los pesos o no, para así volver a empezar desde la primera entrada en otra iteración, o seguir con la siguiente.



```
    # Valores de entrada
    def mostrarValoresIniciales(self):
        print(f"Patrones de entrada: \n{entradas}")
        print(f"Patrones de salida esperados: \n{salidas}")
        print("Peso inicial:", self.pesos)
        print("Peso del Bias:", self.bias)

    #Valores de salida
    def mostrarValoresFinales(self):
        print("Pesos finales:", self.pesos)
        print("Peso del Bias final:", self.bias)
```

Imagen 3: Funciones para mostrar el valores iniciales y finales

Fuente: Elaboración propia

Las funciones mostrarValoresIniciales y mostrarValoresFinales, se utilizarán para depurar y verificar el estado del PS antes y después de su entrenamiento.

```python
if __name__ == "__main__":

    #definición de las combinaciones de las posibilidades
    entradas = [[1, 1, 1],
                [1, 1, -1],
                [1, -1, 1],
                [1, -1, -1],
                [-1, 1, 1],
                [-1, 1, -1],
                [-1, -1, 1],
                [-1, -1, -1]]

    #definición de las posibles salidas del perceptrón
    salidas = [1, 1, 1, 1, 1, 1, 1, -1]

    # Inicio del entrenamiento
    perceptron = PerceptronPaloma()

    #Mostrar valores iniciales
    perceptron.mostrarValoresIniciales()

    # Entrenar el perceptrón
    errores_epoca = perceptron.entrenar(entradas, salidas)
    print("")

    # Mostrar pesos y bias finales
    perceptron.mostrarValoresFinales()

    # Gráfica de la disminución del error
    plt.plot(errores_epoca)
    plt.title('Error por épocas en entremamiento')
    plt.xlabel('Épocas')
    plt.ylabel('Error total')
    plt.show()
```

Imagen 4: Función principal

Fuente: Elaboración propia

Se definen las variables de entrada y de salida en base a la tabla de combinaciones. Luego se genera un objeto de tipo *PerceptronPaloma*, el cual invoca al método de mostrar los valores iniciales, para presentar el estado del perceptrón en un estado inicial, y luego se llama al método para entrenarlo, que a su vez guarda en una variable los resultados que retornan; esta variable se encarga de guardar todos los errores que serán representados con la función plot. de la librería *matplotlib.*

1.2.2 Gráficos

Dependiendo las pruebas que se realicen, la gráfica mostrará el comportamiento del PS en un lapso de 100 épocas máximas; para ver mejor los resultados del entrenamiento y determinar su eficacia, se realizarán 3 pruebas para así compararlas y obtener conclusiones.

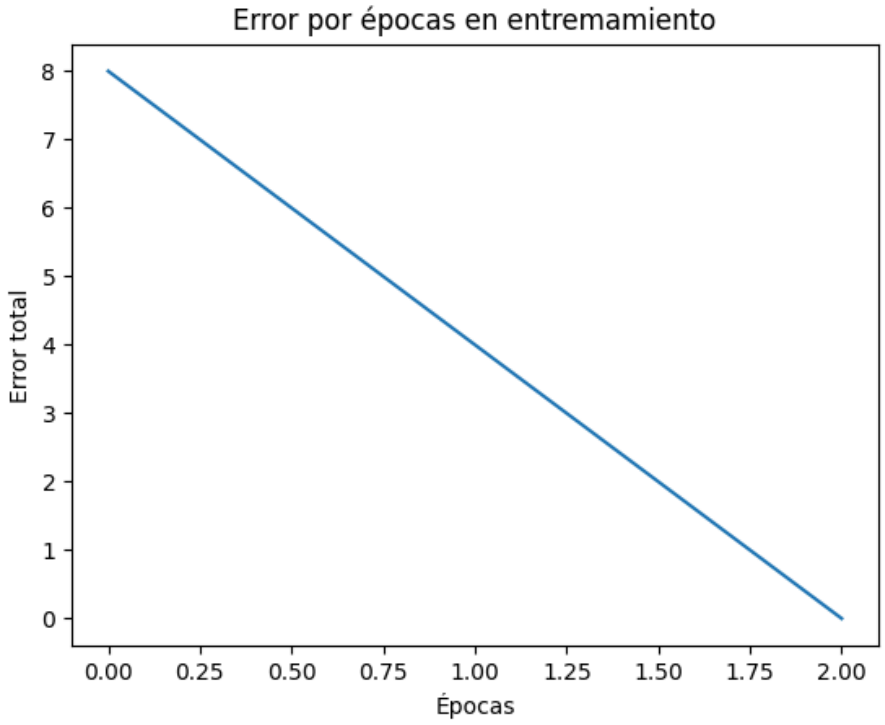


Imagen 5: Gráfica de la prueba 1

Fuente: Elaboración propia

```
Patrones de entrada:
[[1, 1, 1], [1, 1, -1], [1, -1, 1], [1, -1, -1], [-1, 1, 1], [-1, 1, -1], [-1, -1, 1], [-1, -1, -1]]
Patrones de salida esperados:
[1, 1, 1, 1, 1, 1, 1, -1]
Peso inicial: [-0.8475359981962447, 0.211656475979074, 0.5734361498220746]
Peso del Bias: 0.3260209832250839

Pesos finales: [0.352464   0.21165648 0.57343615]
Peso del Bias final: 0.7260209832250839
```

Imagen 6: Resultados por consola de la prueba 1

Fuente: Elaboración propia

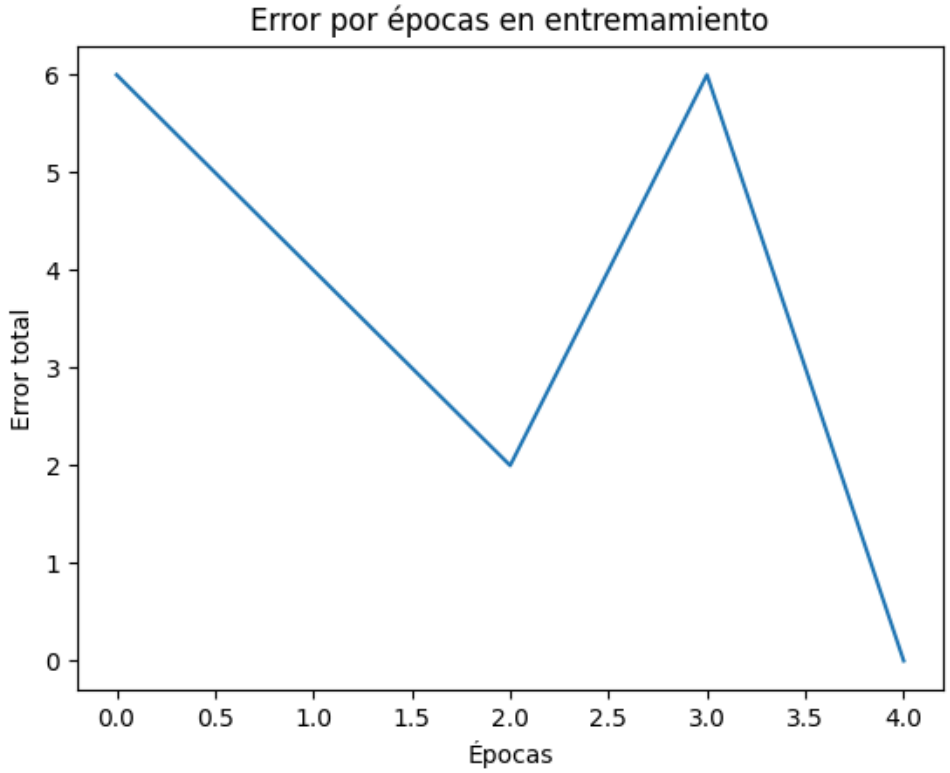


Imagen 7: Gráfica de la prueba 2

Fuente: Elaboración propia

```
Patrones de entrada:
[[1, 1, 1], [1, 1, -1], [1, -1, 1], [1, -1, -1], [-1, 1, 1], [-1, 1, -1], [-1, -1, 1], [-1, -1, -1]]
Patrones de salida esperados:
[1, 1, 1, 1, 1, 1, 1, -1]
Peso inicial: [-0.8775185180335174, -0.4868010244034149, 0.48429029557861547]
Peso del Bias: 0.8070245740027268

Pesos finales: [0.52248148 0.51319898 0.6842903 ]
Peso del Bias final: 1.0070245740027268
```

Imagen 8: Resultado por consola de la prueba 2

Fuente: Elaboración propia

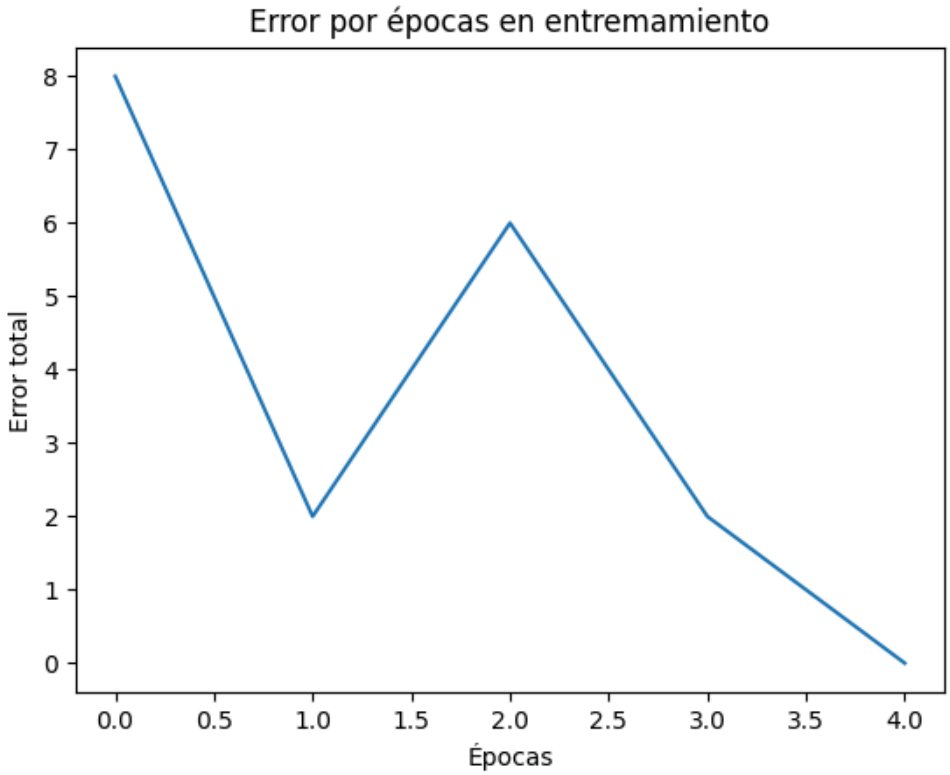


Imagen 9: Gráfica de la prueba 3

Fuente: Elaboración propia

```
Patrones de entrada:
[[1, 1, 1], [1, 1, -1], [1, -1, 1], [1, -1, -1], [-1, 1, 1], [-1, 1, -1], [-1, -1, 1], [-1, -1, -1]]
Patrones de salida esperados:
[1, 1, 1, 1, 1, 1, 1, -1]
Peso inicial: [0.3019296386649113, -0.3573288390262752, -0.6814534441925582]
Peso del Bias: 0.1840996916378086

Pesos finales: [0.50192964 0.24267116 0.31854656]
Peso del Bias final: 0.7840996916378087
```

Imagen 10: Resultado por consola de la prueba 3

Fuente: Elaboración propia

1.3 Análisis del resultado

En las tres pruebas realizadas se pueden ver comportamientos distintos; en la primera da a entender que el PS pudo entrenar de forma eficiente, pues en un lapso aproximado de 2 épocas, pudo disminuir de manera lineal y significativa su error hasta llegar al valor de 0, lo que se entiende que entrenó de manera satisfactoria.

En la segunda prueba se presenta una situación en la que cuando va por la segunda época, vuelve a fallar completamente, y reinicia el entrenamiento desde el inicio para luego completarlo satisfactoriamente de manera lineal en la época 4.

Por último en la tercera prueba, se puede ver cómo inicialmente el PS aprendía de manera rápida en una (1) época para luego volver a fallar un poco, y retomar el aprendizaje en una época, teniendo una leve tardanza en la época 3 donde finaliza su aprendizaje en la época 4.

Conclusiones

El Perceptrón muestra algunas fallas a la hora de ejecutarlo, puesto que en algunas pruebas el aprendizaje puede ser rápido, y en otras tardado; estas fallas son producto de la aleatoriedad de los pesos y su reasignación, las cuales en base a la lógica del algoritmo, es propenso a que falle y genere tiempos de procesamiento prolongados mientras prueba valores óptimos. En algunas de las pruebas realizadas se puede ver cómo los pesos cambian desde valores positivos a negativos o viceversa, todo esto con el fin de hallar una configuración ideal.

De esta manera, se puede decir que el problema de la paloma ha sido solucionado, permitiendo que esta pueda obtener su alimento; aunque teniendo presente que de 8 posibilidades, en 7 lo puede obtener y en 1 no, la capacidad de aprendizaje debería ser alta y con un error mínimo, ya que las condiciones de éxitos predominan ante la de fallo.

2. Problema 2: Diagnostico médico

2.1 Descripción del problema

El objetivo es diseñar un perceptrón simple que pueda diagnosticar si un paciente tiene cierta enfermedad o no en función de varios síntomas.

Si una persona posee dos síntomas se califica como enfermo, si posee solo un síntoma se califica como sano.

Entrada (tres entradas):

- Fiebre (-1 si no tiene fiebre, 1 si tiene fiebre).
- Dolor de cabeza (-1 si no tiene dolor de cabeza, 1 si tiene dolor de cabeza).
- Fatiga (-1 si no tiene fatiga, 1 si tiene fatiga).

Salida (una salida):

- 1 si el paciente tiene la enfermedad.
- -1 si el paciente no tiene la enfermedad.

2.2 Propuesta de la solución

Primero se define la tabla que modela las distintas posibilidades del perceptrón, en este caso, el contexto que determina cuando un paciente es considerado sano o enfermo.

X1	X2	X3	ENFERMO
1	1	1	1
1	1	-1	1
1	-1	1	1
1	-1	-1	-1
-1	1	1	1
-1	1	-1	-1
-1	-1	1	-1
-1	-1	-1	-1

Tabla 2: Posibilidades para diagnosticar a un paciente

Fuente: Elaboración propia

Se puede concluir que las posibles situaciones son simétricas, puesto que guardan relación reflexiva, donde si una persona cuenta con 3 o 2 síntomas, se considerará enferma, mientras que cuando solo tenga 1 síntoma, se considera sana; por otro lado, cuando no cuenta con 2 o 3 síntomas se considera sana, excepto cuando no cuenta con 1 uno, que se determina como enfermo.

Por otro lado, mediante el lenguaje Python se aprovechó las librerías *numpy*, y *matplotlib*, para hacer más práctico y eficiente el algoritmo.

2.2.1 Implementación del algoritmo

El algoritmo está compuesto por una clase para representar el perceptrón; y diferentes métodos para realizar las funciones de activación, el entrenamiento de la red, y la impresión de los valores por pantalla.

```
import random
import numpy as np
import matplotlib.pyplot as plt

class PerceptronMedico:
    def __init__(self):
        self.pesos = [random.uniform(-1, 1) for _ in range(3)] # valores aleatorios para el p
        self.bias = random.uniform(0, 1) # valores aleatorios para el peso del bias entre 0 y
        self.tasaAprendizaje = 0.1 # valor de la tasa de aprendizaje
```

Imagen 11: Definición del Perceptrón que modela el diagnóstico médico
Fuente: Elaboración propia

Mediante el uso de la librería *numpy* y la librería *random*, se utilizan funciones para definir valores aleatorios a cada una de las variables de entrada (3 en total), y el bias; además, se define un valor constante para la tasa de aprendizaje, de 0.1, el cual ayudará a variar en menor grado la actualización de los pesos y bias.

```python
# funcion de activación
def funcionActivacion(self, suma):
    return 1 if suma > 0 else -1

# logica de entrenamiento del perceptron
def entrenar(self, inputs, desired_outputs):
    errores_epoca = []
    for iteraciones in range(100):  # Número de épocas de entrenamiento
        error = False
        error_total = 0
        for i in range(len(inputs)):
            suma = np.dot(inputs[i], self.pesos) + self.bias
            output = self.funcionActivacion(suma)
            error = error or (output != desired_outputs[i])
            error_term = desired_outputs[i] - output

            self.pesos += self.tasaAprendizaje * error_term * np.array(inputs[i])
            self.bias += self.tasaAprendizaje * error_term

            error_total +=abs(error_term)
        errores_epoca.append(error_total)

        if not error:
            break
    return errores_epoca
```

Imagen 12: Función de activación y de entrenamiento

Fuente: Elaboración propia

La función de entrenamiento se encargará analizar todas las posibilidades buscando que dependiendo las condiciones de entrada, esta pueda acertar con el resultado; para esto recibe por parámetro una lista de variables de entrada, y las salidas esperadas. Posteriormente inicializa una lista para guardar los errores encontrados y más adelante generar una gráfica a partir de ella; el entrenamiento se dará en un límite finito de 100 épocas, donde el perceptrón puede aprender en una iteración menor o igual a 100.

El entrenamiento se basa en calcular las variables de entrada con sus respectivos pesos, y el bias, para luego evaluar ese valor mediante la función de activación hardlim, donde si el valor es mayor a 0, toma un 1, o en caso contrario, un .1; en base a ese resultado se determina si para esa entrada que se está evaluando es

necesario recalcular los pesos o no, para así volver a empezar desde la primera entrada en otra iteración, o seguir con la siguiente.

```python
# valores de entrada
def mostrarValoresIniciales(self):
    print(f"Patrones de entrada: \n{entradas}")
    print(f"Patrones de salida esperados: \n{salidas}")
    print("Peso inicial:", self.pesos)
    print("Peso del Bias:", self.bias)

# valores de salida
def mostrarValoresFinales(self):
    print("Pesos finales:", self.pesos)
    print("Peso del Bias final:", self.bias)
```

Imagen 13: Funciones para mostrar el valores iniciales y finales

Fuente: Elaboración propia

Las funciones mostrarValoresIniciales y mostrarValoresFinales, se utilizarán para depurar y verificar el estado del PS antes y después de su entrenamiento.

```python
if __name__ == "__main__":
    #definición de las combinaciones de las posibilidades
    entradas = [[1, 1, 1],
                [1, 1, -1],
                [1, -1, 1],
                [1, -1, -1],
                [-1, 1, 1],
                [-1, 1, -1],
                [-1, -1, 1],
                [-1, -1, -1]]

    #definición de las posibles salidas del perceptrón
    salidas = [1, 1, 1, -1, 1, -1, -1, -1]

    # Inicio del entrenamiento
    perceptron = PerceptronMedico()

    #Mostrar valores iniciales
    perceptron.mostrarValoresIniciales()

    # Entrenar el perceptrón
    errores_epoca = perceptron.entrenar(entradas, salidas)
    print("")

    # Mostrar pesos y bias finales
    perceptron.mostrarValoresFinales()

    # Gráfica de la disminución del error
    plt.plot(errores_epoca)
    plt.title('Error por épocas en entrenamiento')
    plt.xlabel('Épocas')
    plt.ylabel('Error total')
    plt.show()
```

Imagen 14: Función principal

Fuente: Elaboración propia

Se definen las variables de entrada y de salida en base a la tabla de combinaciones. Luego se genera un objeto de tipo *PerceptronMedico*, el cual invoca al método de mostrar los valores iniciales, para presentar el estado del perceptrón en un estado inicial, y luego se llama al método para entrenarlo, que a su vez guarda en una variable los resultados que retornan; esta variable se encarga de guardar todos los errores que serán representados con la función plot. de la librería *matplotlib*

2.2.2 Gráficos

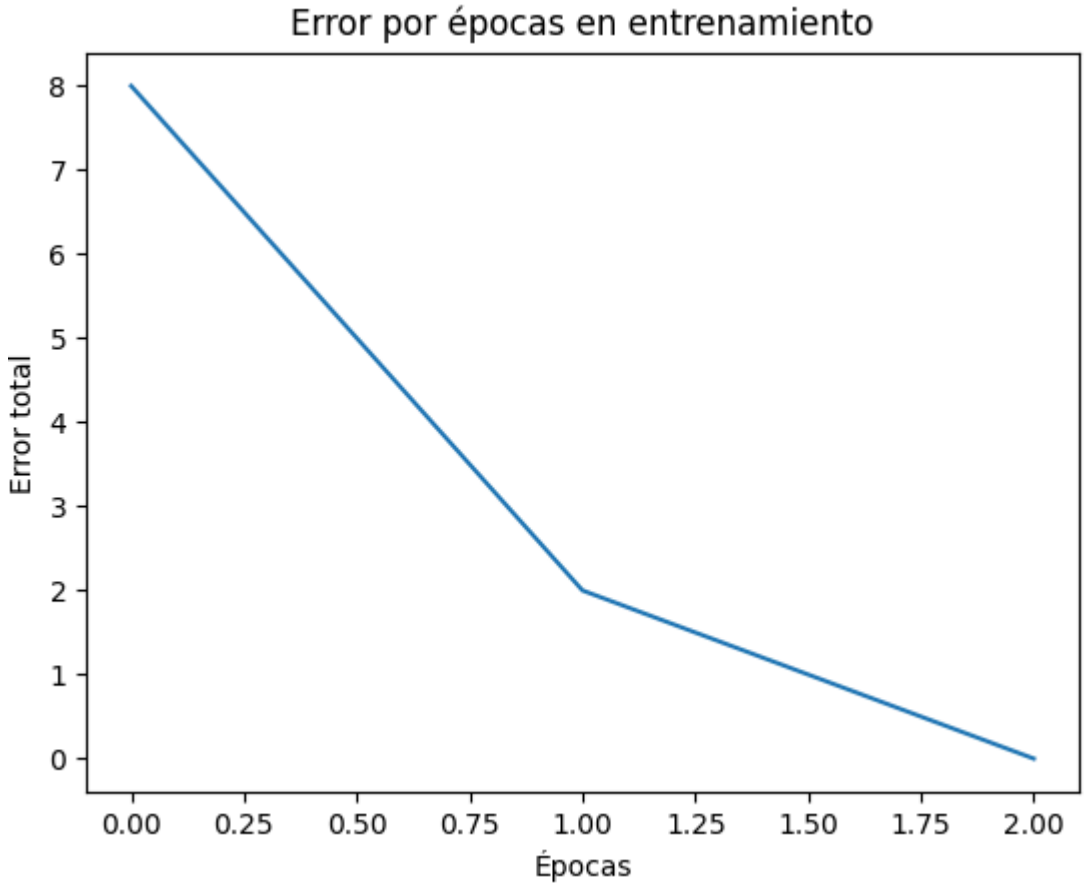


Imagen 15: Gráfico de la prueba 1

Fuente: Elaboración propia

```
Patrones de entrada:
[[1, 1, 1], [1, 1, -1], [1, -1, 1], [1, -1, -1], [-1, 1, 1], [-1, 1, -1], [-1, -1, 1], [-1, -1, -1]]
Patrones de salida esperados:
[1, 1, 1, -1, 1, -1, -1, -1]
Peso inicial: [0.12106238735010422, 0.15354656283861634, -0.7405645547806781]
Peso del Bias: 0.18219672925289265

Pesos finales: [0.32106239 0.35354656 0.25943545]
Peso del Bias final: -0.017803270747107358
```

Imagen 16: Resultados por consola de la prueba 1

Fuente: Elaboración propia

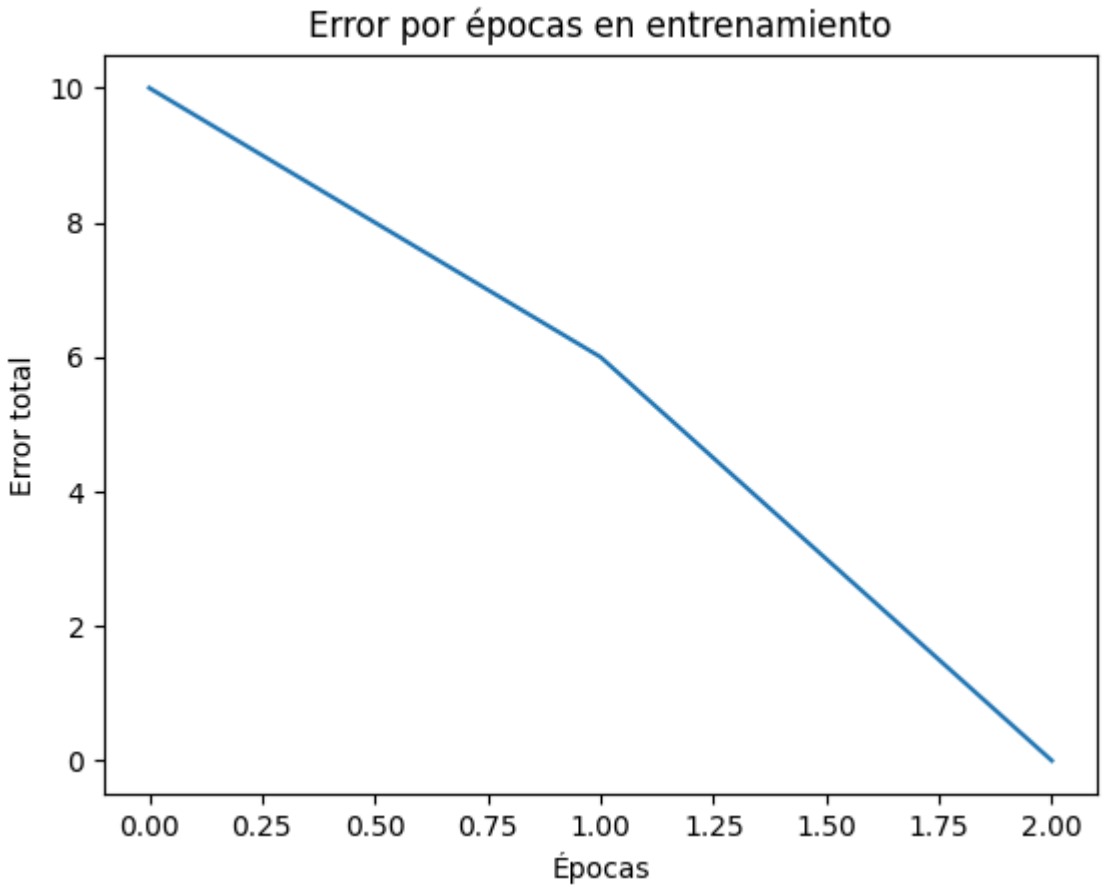


Imagen 17: Gráfico de la prueba 2

Fuente: Elaboración propia

```
Patrones de entrada:
[[1, 1, 1], [1, 1, -1], [1, -1, 1], [1, -1, -1], [-1, 1, 1], [-1, 1, -1], [-1, -1, 1], [-1, -1, -1]]
Patrones de salida esperados:
[1, 1, 1, -1, 1, -1, -1, -1]
Peso inicial: [-0.37976864556706413, 0.4666735593076241, -0.5127295340837079]
Peso del Bias: 0.9061300877025659

Pesos finales: [0.42023135 0.46667356 0.68727047]
Peso del Bias final: 0.10613008770256604
```

Imagen 18: Resultados por consola de la prueba 2

Fuente: Elaboración propia

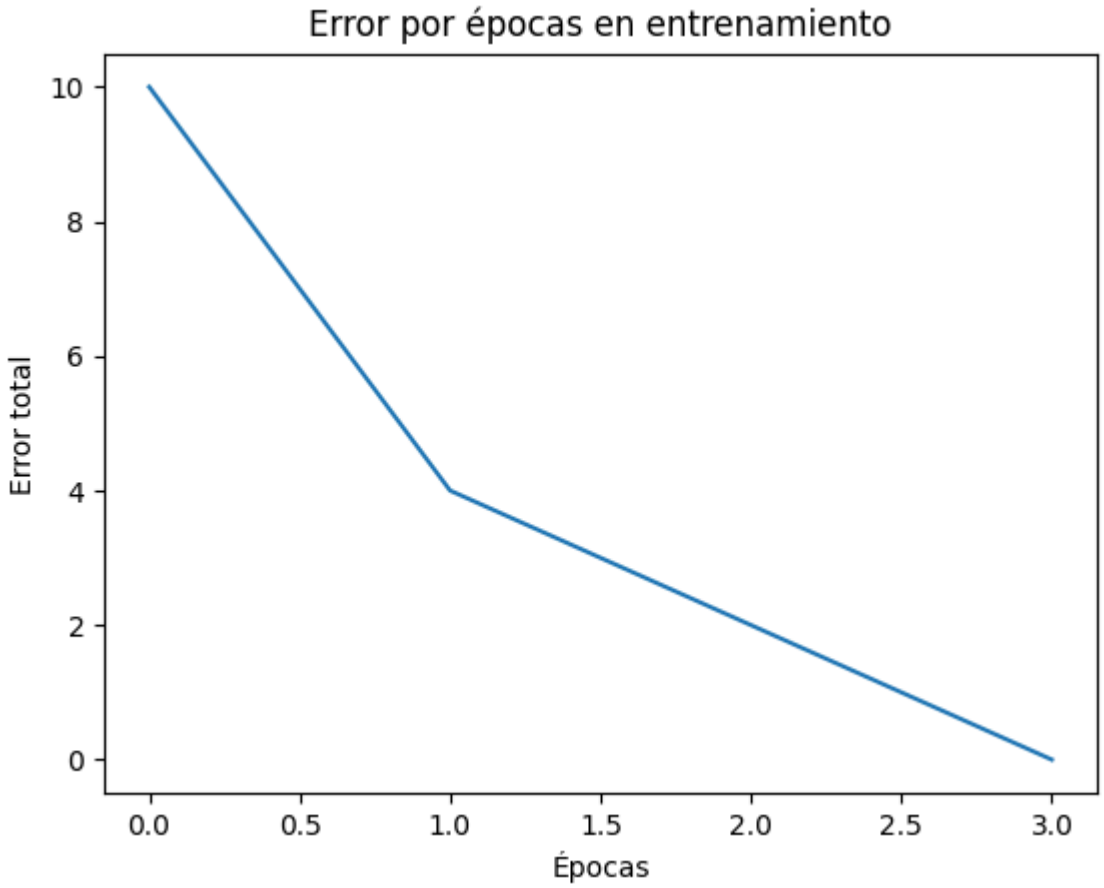


Imagen 19: Gráfico de la prueba 3

Fuente: Elaboración propia

```
Patrones de entrada:
[[1, 1, 1], [1, 1, -1], [1, -1, 1], [1, -1, -1], [-1, 1, 1], [-1, 1, -1], [-1, -1, 1], [-1, -1, -1]]
Patrones de salida esperados:
[1, 1, 1, -1, 1, -1, -1, -1]
Peso inicial: [-0.8059324381461053, 0.7635760149513646, -0.9242581277787505]
Peso del Bias: 0.42262320849555846

Pesos finales: [0.39406756 0.36357601 0.67574187]
Peso del Bias final: 0.02262320849555849
```

Imagen 20: Resultado por consola de la prueba 3

Fuente: Elaboración propia

2.3 Análisis del resultado

A diferencia del problema 1, las tres pruebas realizadas se pueden ver comportamientos similares, en donde tienen en común realizar el entrenamiento, y luego llegar a un punto donde, o tarda un poco más, o un poco menos, completar el aprendizaje. En la primera situación se puede ver cómo el PS aprende de manera rápida de a una unidad de error, en cambio al llegar a la época 1, disminuye un poco su aprendizaje lo que hace que el cambio de error disminuya a la mitad; sin embargo, a pesar de ello, logra terminar lo poco que le faltaba para alcanzar un error de 0.

Para la segunda prueba se logra apreciar que su entrenamiento es casi completamente lineal, exceptuando que tuvo un momento en la época 1 donde tuvo una leve mejoría en su comportamiento lo que le permitió alcanzar el error de 0.

Y para la última prueba, es una donde su cambio en el comportamiento es notorio, donde de manera temprana alcanza un punto en la época 1, donde se la tasa de entrenamiento disminuye un poco y hace que finalice satisfactoriamente en una época 3.

Conclusiones

Teniendo en cuenta los resultados obtenidos, este segundo perceptrón parece tener una mejora considerable a comparación del primero, en donde era más probable que el error aumente, y deba empezar desde el inicio, algo que no sucede para el problema del diagnóstico médico. Sin embargo, ambos están diseñados de la misma manera, lo cual debería mantener similitud en su lógica, pero posiblemente lo que genera el cambio es su patrón de salida; pues, a pesar de que el primero tiene un contexto aparentemente más fácil, el segundo guarda una reflexión en los datos, lo cual hace que al aprender un primer grupo de situaciones, el segundo se sabe obteniendo los valores opuestos del primero.