

Un perceptrón es un tipo de red neuronal artificial que se utiliza para tareas de clasificación binaria. **Entradas:** El perceptrón recibe múltiples entradas, cada una con un peso asociado. **Pesos:** Los pesos determinan la importancia de cada entrada. **Sesgo (Bias):** Se añade un término de sesgo para ajustar la salida del modelo. **Función de Activación:** Una función que decide si la neurona se activa o no. **Inicialización:** Los pesos y el sesgo se inicializan, generalmente con valores pequeños o ceros. **Cálculo de la Salida:** Se calcula una combinación lineal de las entradas y los pesos. Se añade el sesgo. La combinación resultante se pasa a través de la función de activación (por ejemplo, una función escalón). Se ajustan los pesos y el sesgo utilizando un algoritmo de aprendizaje supervisado, como la regla delta. El objetivo es minimizar el error entre la salida predicha y la salida deseada. Este proceso se repite durante varias épocas hasta que el modelo converge. **Ejemplo** Para una compuerta lógica AND, el perceptrón aprenderá a producir una salida de 1 solo cuando ambas entradas sean 1. Para una compuerta OR, producirá una salida de 1 si al menos una de las entradas es 1.

En esta parte del código como se nos pidió. Experimentar con el Código proporcionado de la implementación de la compuerta OR en la presentación Clase 1.

Generar
create a dataframe with 2 columns and 10 rows
Cerrar

```

import numpy as np

class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, epochs=1000):
        self.W = np.zeros(input_size + 1) # +1 for the bias term
        self.learning_rate = learning_rate
        self.epochs = epochs

    def activation_function(self, x):
        return 1 if x >= 0 else 0

    def predict(self, x):
        x = np.insert(x, 0, 1) # Insert bias term
        z = self.W.T.dot(x)
        return self.activation_function(z)

    def train(self, X, y):
        for _ in range(self.epochs):
            for inputs, label in zip(X, y):
                prediction = self.predict(inputs)
                self.W += self.learning_rate * (label - prediction) * np.insert(inputs, 0, 1)

    def print_weights(self):
        print(f"Pesos después del entrenamiento: {self.W}")

# Datos de entrada y salidas deseadas para la compuerta OR
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y = np.array([0, 1, 1, 1])

# Crear el perceptron y entrenarlo
perceptron = Perceptron(input_size=2)
perceptron.train(X, y)

# Imprimir los pesos después del entrenamiento
perceptron.print_weights()

# Probar el perceptron
for inputs in X:
    print(f"Entrada: {inputs} > Salida predicha: {perceptron.predict(inputs)}")

```

Pesos después del entrenamiento: [-0.1 0.1 0.1]
Entrada: [0 0] > Salida predicha: 0
Entrada: [0 1] > Salida predicha: 1
Entrada: [1 0] > Salida predicha: 1
Entrada: [1 1] > Salida predicha: 1

A continuación se implementa lo mismo pero con AND. realizar la implementación de la compuerta AND. Hacer la comprobación manual e identificar qué pasa si modificamos los hiperparametros de la red (learning rate y el número de épocas).

```

import numpy as np

class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, epochs=1000):
        self.W = np.zeros(input_size + 1) # +1 for the bias term
        self.learning_rate = learning_rate
        self.epochs = epochs

    def activation_function(self, x):
        return 1 if x >= 0 else 0

    def predict(self, x):
        x = np.insert(x, 0, 1) # Insert bias term
        z = self.W.T.dot(x)
        return self.activation_function(z)

    def train(self, X, y):
        for _ in range(self.epochs):
            for inputs, label in zip(X, y):
                prediction = self.predict(inputs)
                self.W += self.learning_rate * (label - prediction) * np.insert(inputs, 0, 1)

    def print_weights(self):
        print(f"Pesos después del entrenamiento: {self.W}")

# Datos de entrada y salidas deseadas para la compuerta AND
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y = np.array([0, 0, 0, 1])

# Crear el perceptron y entrenarlo
perceptron = Perceptron(input_size=2)
perceptron.train(X, y)

# Imprimir los pesos después del entrenamiento
perceptron.print_weights()

# Probar el perceptron
for inputs in X:
    print(f"Entrada: {inputs} > Salida predicha: {perceptron.predict(inputs)}")

```

```

↔ Pesos después del entrenamiento: [-0.2  0.2  0.1]
Entrada: [0 0] > Salida predicha: 0
Entrada: [0 1] > Salida predicha: 0
Entrada: [1 0] > Salida predicha: 0
Entrada: [1 1] > Salida predicha: 1

```

```

# Datos de entrada y salidas deseadas para la compuerta AND
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y = np.array([0, 0, 0, 1])

```

## Para la compuerta XOR

```

# Datos de entrada y salidas deseadas para la compuerta XOR
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y = np.array([0, 1, 1, 0])

```

un perceptrón simple puede resolver problemas linealmente separables como AND, pero no puede resolver problemas no linealmente separables como XOR. Para resolver XOR, es necesario una red neuronal con al menos una capa oculta. Como se explico en clase.

The first of these is the fact that the system is not a simple one. It is a complex system, and as such, it is not possible to understand it by looking at its parts in isolation. The system is a whole, and its behavior is determined by the interactions between its parts. This is a fundamental principle of systems thinking, and it is one that is often overlooked in traditional approaches to problem-solving.

The second of these is the fact that the system is dynamic. It is not a static system, and its behavior changes over time. This is another fundamental principle of systems thinking, and it is one that is often overlooked in traditional approaches to problem-solving.

The third of these is the fact that the system is open. It is not a closed system, and it interacts with its environment. This is another fundamental principle of systems thinking, and it is one that is often overlooked in traditional approaches to problem-solving.

The fourth of these is the fact that the system is self-organizing. It is not a system that is imposed from the outside, but one that emerges from the interactions between its parts. This is another fundamental principle of systems thinking, and it is one that is often overlooked in traditional approaches to problem-solving.

The fifth of these is the fact that the system is resilient. It is not a system that is fragile and easily broken, but one that is able to withstand change and maintain its essential characteristics. This is another fundamental principle of systems thinking, and it is one that is often overlooked in traditional approaches to problem-solving.

The sixth of these is the fact that the system is sustainable. It is not a system that is designed to last for a short time, but one that is designed to last for a long time. This is another fundamental principle of systems thinking, and it is one that is often overlooked in traditional approaches to problem-solving.

The seventh of these is the fact that the system is equitable. It is not a system that is designed to benefit a few at the expense of many, but one that is designed to benefit all. This is another fundamental principle of systems thinking, and it is one that is often overlooked in traditional approaches to problem-solving.

The eighth of these is the fact that the system is just. It is not a system that is designed to be unfair, but one that is designed to be fair. This is another fundamental principle of systems thinking, and it is one that is often overlooked in traditional approaches to problem-solving.

The ninth of these is the fact that the system is transparent. It is not a system that is designed to be opaque, but one that is designed to be clear. This is another fundamental principle of systems thinking, and it is one that is often overlooked in traditional approaches to problem-solving.

The tenth of these is the fact that the system is accountable. It is not a system that is designed to be unaccountable, but one that is designed to be responsible. This is another fundamental principle of systems thinking, and it is one that is often overlooked in traditional approaches to problem-solving.